碩士學位論文

리눅스 파일시스템 개발을 위한 템플릿 구성에 관한 연구

2007 年

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

장 명 우

碩 士 學 位 論 文
指導敎授 李 珉 秔

리눅스 파일시스템 개발을 위한 템플릿 구성에 관한 연구

Research on the template for development of the Filesystem in Linux

2007年 6月 日

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

장 명 우

碩 士 學 位 論 文
指導敎授 李 珉 秔

리눅스 파일시스템 개발을 위한 템플릿 구성에 관한 연구

Research on the template for development of the Filesystem in Linux

위 論文을 컴퓨터工學 碩士學位論文으로 提出함

2007年 6月 日

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

장 명 우

장명우의 工學 碩士學位論文을 인정함

2007年 6月 日

심사위원장 김 영 웅 (인)

심사위원 이 민 석 (인)

심사위원 김 성 동 (인)

목 차

표 목 차

그 림 목 차

요 약

파일시스템은 데이터를 저장, 검색 할 수 있는 운영체제의 부 시스템이다. 전통적으로 파일시스템은 범용으로 제작되는 경우가 많으나 최근에는 다양한 멀티미디어 응용 제품들의 출현으로 특정 목적에 특화된 파일시스템이 필요하다. 오픈소스로 개발되고 있는 운영제제 중의 하나인 리눅스는 파일시스템의 추상 계층을 제공하여 다양한 파일시스템을 지원하다. 이 추상 모델이 새로운 파일시스템의 지원을 쉽게 하지만 구현적인 관점에서는 부족한 관련 정보 때문에 기술 장벽이 높다.

본 논문에서는 파일시스템 개발의 기술 장벽을 낮추기 위하여 관련 정보가 필요하다는 것을 인식하여 그 정보를 제공하고 개발 방법을 제안하였다. 리눅스 파일시스템의 분석을 통해 파일시스템 구현 시 필요한 정보를 문서화하고 파일시스템 템플릿을 구현하였다. 문서와 템플릿을 이용하여 파일시스템 개발을 목표로 하는 연구자들이 개발의 생산성을 높일 수 있게 하고 불필요한 노력을 줄일 수 있다. 문서에는 리눅스 파일시스템의 설명과 각종 API 등을 작성하여 개발 시 참고할 수 있고 파일시스템 템플릿을 사용하면 개발을 보다 상위 단계에서 시작할 수 있다.

제 1 장 서론

1.1 연구 동기

파일시스템(filesystem)은 운영체제의 발전과 함께 그 일부로서 끊임없이 발전해 오고 있다. 파일시스템은 그 이름 그대로 파일을 관리하는 시스템이며, 운영체제가 다양한 저장장치로부터 데이터를 저장하고 검색할 수 있게 한다[1]. 파일시스템은 운영체제의 서브시스템으로서 중요할 뿐만 아니라, 파일시스템 자체의 성능은 CPU에 상대적으로 느린 저장 장치(예. 하드디스크)들과 상호 동작하기 때문에 시스템 전체 성능에 지대한 영향을 미치는 요소가 된다. 현대에는 다양한 멀티미디어 디바이스들의 수요와 생산이 나날이 증대됨에 따라 파일시스템 역시 다양한 목적으로 개발이 이루어지고 있다. 하지만, 그 개발이 복잡하고 어려움이 있고 파일시스템의 기본적인 요소에 대한 중복된 노력이 반복되고 있다.

한편, 리눅스(Linux)에서는 다양한 파일시스템을 지원하기 위해 파일시스템들의 추상 계층인 Virtual Filesystem(이하 VFS)를 제공한다. 이로 인해 파일시스템의 개발이 용이하게 되었지만, 관련 문서의 부재 등 여러 가지 이유로 리눅스에서의 파일시스템 개발에는 여전히 높은 기술 장벽이 있다.

리눅스에서 새로운 파일시스템을 개발하거나 이식(porting)하기 위해서는 VFS와 개발될 파일시스템과의 상호 관계를 명확하게 인지하고 있어야만 한다. 주목할 점은 리눅스의 파일시스템에서는 기본적인 기능들은 대부분 유사하고 거기에 특정 파일시스템만의 기능이 추가되어 제작되는 것이다. 이 기본적인 기능들을 구현하기 위해서 많은 시간과 인력이 낭비되고 있다. 만일 VFS와 완벽한 호환이 되고 파일시스템의 기본적인 기능이 구현되어 있는 파일시스템이 존재한다면, 파일시스템 개발에 있어서 많은 시

간과 노력을 줄일 수 있고, 좀 더 상위 단계에서 개발을 시작할 수 있다. 즉, 파일시스템이 초점을 맞춰야 할 블록의 할당 전략, 디렉터리 관리 방법 등에 더 많은 노력을 투자할 수 있고 리눅스 내에서 동작하기 위한 여러 가지 구현을 쉽게 할 수 있게 된다. 이러한 관점에서 본 논문에서 VFS와 특정 파일시스템과의 상호 작용을 면밀히 분석하여 간단한 디스크 레이아웃을 가지고 특정 파일시스템에 의존한 부분을 배제, 분리 하여 이해가 쉬우면서도 완전히 동작하며 신뢰성 있는 파일시스템을 개발하여 각종 테스트를 통해 검증하고, 리눅스에서 다양한 파일시스템 개발을 용이하게 할 수 있는 리눅스 파일시스템 코드 템플릿과 그 개발방법을 제안한다.

1.2 연구 목적

본 논문은 리눅스에서 파일시스템 개발에 필요한 모든 정보를 제공하고 그 개발의 시작점이 될 수 있는 파일시스템 템플릿을 제공하는 데 그 목적이 있다. 다음의 단계별 목표를 성취함으로써 그 목적을 달성한다. 이 연구를 통한 산출물은 파일시스템 개발에 관련된 문서 즉 논문 자체와 파일시스템 템플릿과 그것에 관련한 유틸리티들이다.

- **VFS의 요구사항 분석**

  VFS은 유닉스(UNIX)의 파일 공통 모델을 따르기 때문에 각 파일시스템은 VFS에서 요구하는 객체와 메쏘드들을 구현해야 한다. 그러므로 VFS의 요구사항을 Source Code를 통해 면밀히 분석하여 파일시스템이 VFS 하위 모듈로 동작하기 위해 요구되는 필수적인 요소가 무엇인지 명세화 한다.

- **파일시스템 의존적인 부분 분리**

  특정 파일시스템에 의존적인 부분과 공통적으로 적용되는 부분을 분리하여 구현을 용이하게 한다. 공통적인 부분의 의미는 파일시스템 코드에서 VFS의 인터페이스에 맞게 동작하여야 할 부분들을 의미하며 이것은 리눅스에서 동작하는 모든 파일시스템들이 공통적으로 구현하여야 한다. 이 분리 작업을 통해 파일시스템 개발 시 공통의 부분을 거의 그대로 적용하여 개발 생산성을 높일 수 있게 한다.

- **파일시스템 템플릿의 개발**

  위의 2가지 연구의 결과를 통해 실제 리눅스 VFS의 하위 모듈로 동작하는 파일시스템을 개발한다. 파일시스템은 완전한 동작을 해야 한다. 즉 리눅스 표준 시스템 콜들의 요청에 대한 적합한 처리를 하는 것을 의미한다. 소스 코드 자체가 문서가 되므로 가독성이 좋아야 하고 이해하기가 쉬워야 한다. 유저레벨의 테스트 프로그램으로 파일시스템 관련 함수들의 요청을 제대로 서비스 하는 지를 검증하도록 한다. 이 파일시스템이 파일시스템 템플릿이 된다.

1.3 논문 구성

본 논문의 구성은 다음과 같다.

- 2장 연구 배경

  리눅스 파일시스템과 기존 연구들에 대한 분석과 리눅스에서 파일시스템 개발을 난해하게 하는 요소에 대해 기술한다.

- 3장 연구 내용

  VFS와 리눅스 표준 파일시스템들을 분석하고 그 결과를 토대로 파일시스템을 설계, 구현, 테스트 한다.

- 4장 결과 및 토의

  연구의 결과를 분석하고, 실험의 전체 과정을 통해 도출된 파일시스템 개발 시 본 프레임웍을 적용하는 방법을 제시한다.

- 5장 결론 및 향후 연구

  연구의 결론을 제시하고, 향후 연구는 어떤 목적과 방향이 될지를 기술한다.

제 2 장 연구 배경

2.1 리눅스 가상 파일시스템

2.1.1 리눅스 가상 파일시스템(Virtual Filesystem) 개요

Virtual Filesystem(이하 VFS)는 표준 유닉스 파일 시스템이 제공하는 모든 시스템 콜을 처리하는 커널 소프트웨어 계층이다. 다양한 파일 시스템에 대해 공통의 인터페이스를 제공한다[2].

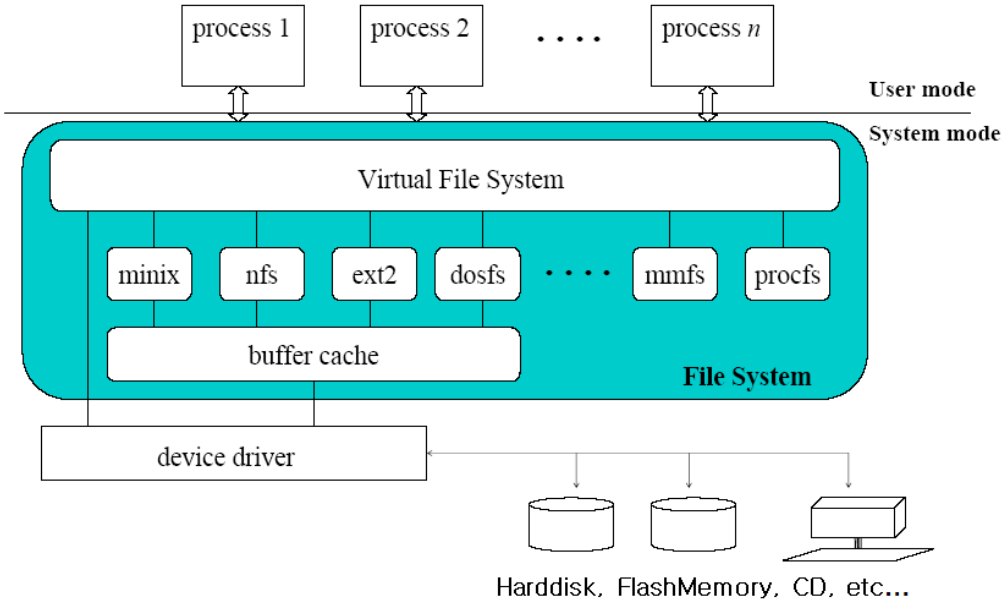

그림 2.1 Virtual Filesystem Overview

그림2.1을 보면, VFS는 User Mode에서 요청한 시스템 콜에 대한 인터페이스이며, 다양한 파일시스템들의 추상계층이 되어, User는 파일시스템의 종류에 무관하게 공통의 시스템 콜을 이용하여 파일시스템을 사용할 수 있다. 즉, 사용자는 dos파일시스템의 파일을 ext2로 복사하려고 할 때 파일시스템의 종류에 무관하게 동일한 시스템 콜을 이용할 수 있음을 의

미한다. 리눅스는 VFS로 인해 다양한 파일시스템 지원이 가능하고, 논문 작성 시의 최신 커널 버전 2.6.20에서는 60여 가지의 파일시스템을 지원한다. 지원이 되는 파일시스템의 종류는 크게 3가지로 나눌 수 있다. 디스크 기반 파일시스템(예: Ex2, FAT32...), 네트웍 파일시스템(예: NFS, Coda, SMB)와 가상 파일시스템이라고 불리우기도 하는 특수 파일시스템이 있다. 본 논문에서는 디스크 기반의 파일시스템에 한해서 논의한다.

VFS는 이렇게 다양한 파일시스템을 지원하면서 I/O 연산에 공통인터페이스를 제공하므로, 사용자 입장에서는 지원이 되는 한 다양한 파일시스템을 호환성을 고려하지 않고도 자유롭게 사용할 수 있고, 파일시스템 개발자 입장에서는 VFS가 파일시스템이 공통적으로 수행하는 많은 부분들을 대신 처리해 주기 때문에 상당량의 코드를 단축시킬 수 있어 개발의 오버헤드를 줄일 수 있다. 그림2.2는 VFS 관련 소스 파일이다.

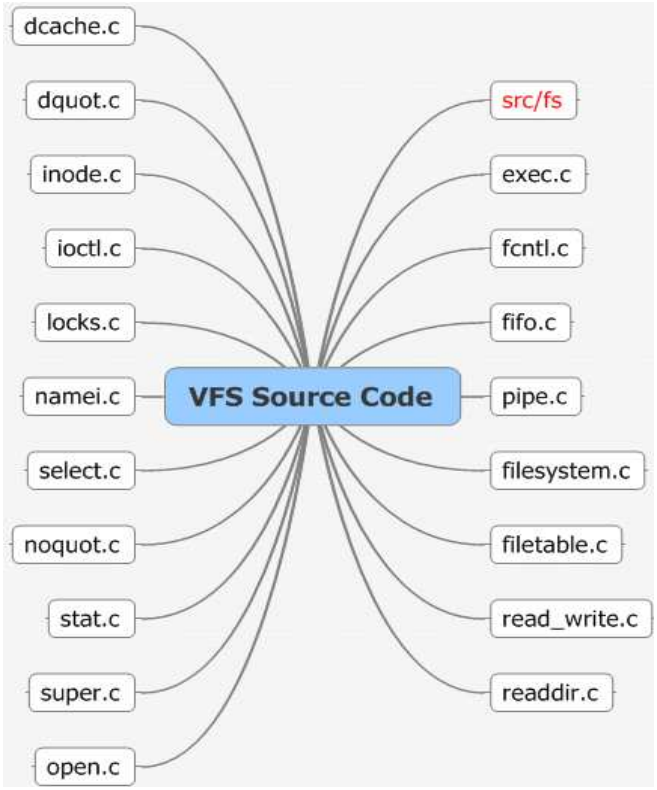

그림 2.2 VFS source tree

2.1.2 VFS Data Structure

VFS는 공통 파일 모델을 지향하며 그것을 위해 4가지의 추상 데이터 타입을 사용한다. VFS는 이 4가지의 추상 객체로 구성되는 객체 지향 프레임웍 이라고 볼 수 있다. 이 객체들은 superblock, inode, dentry, file 이다 [4]. VFS는 OOP(object oriented programming)의 원리로 구현되어 있다. C 언어로 작성되어 있긴 하지만 객체가 메쏘드를 포함하는 함수테이블을 포인팅하는 구조로 되어 있어 동적으로 객체를 할당할 수 있다. 이러한 방식으로 인해 다양한 파일시스템들의 객체를 VFS에서 사용하고 그 연산들을 호출할 수 있게 된다. 객체에서 함수를 포함하는 것이 아니고 함수 테이블을 포인팅 하고 있는 것이 일반적인 객체지향방식의 프로그램과 다르다. 따라서 각 파일시스템들은 그 물리적 구성이 다르지만 이 객체들의 속성과 메쏘드를 구현하면 VFS는 그 객체들을 통해 특정 파일시스템에 I/O 요청을 할 수 있게 된다.

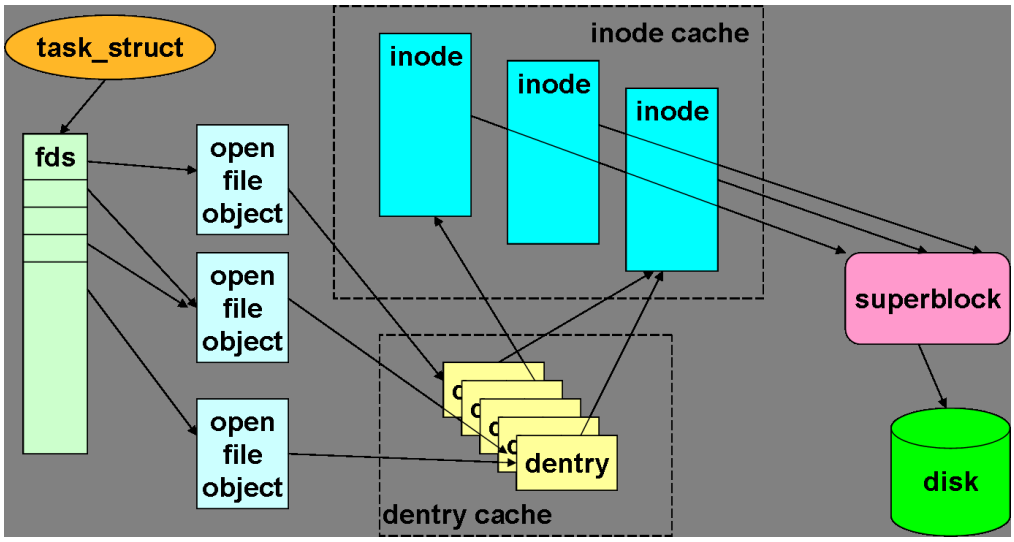

그림 2.3 객체들간의 관계

그림2.3은 각 객체들의 서로 간의 연관성을 나타낸다. 프로세스가 파일을 열면 파일 객체가 생성되고 파일객체는 디렉터리 엔트리를 가르킨다. 디렉터리 엔트리는 그 엔트리에 해당하는 아이노드를 가지고 있다. 아이노드에는 파일시스템 수퍼블록을 포인팅하고 있고 파일시스템 수퍼블록은 실제 디바이스에 관련한 정보를 가지고 있다. 이처럼 긴밀하게 객체들이 연결되어 있다. 유저레벨에서는 파일객체와의 연산만으로 I/O 연산을 하는 것처럼 보이고 나머지는 블랙박스화 되어 있다. 실제 파일시스템에서는 이러한 객체들을 생성, 삭제, 갱신 하는 등의 관리를 한다. VFS 객체들은 특정 파일시스템의 객체들과 일치하지 않더라고 특정 파일시스템에서 변환하여 전달한 정보를 통해 4가지 객체로서 처리하게 된다. 즉 VFS는 파일시스템들도 4가지 객체를 가지고 있는 것처럼 동작하고 파일시스템은 그 동작을 지원하기 위해 각자의 객체들을 VFS의 객체들로 변환하는 작업들을 하게 된다.

가. Super Block Object

수퍼블록 객체는 디스크 기반일 경우 마운트 된(Linux 시스템의 root filesystem의 일부로 올려진) 파일시스템을 제어하는 객체이다. 파일시스템을 제어하는 역할을 하고 복사본을 제외하면 파티션 당 1개가 존재한다.

```
struct super_block {
        struct list_head      s_list;
        dev_t                 s_dev;
        unsigned long         s_blocksize;
        unsigned char         s_blocksize_bits;
        unsigned char         s_dirt;
        unsigned long long    s_maxbytes;
        struct file_system_type       *s_type;
        struct super_operations       *s_op;
        struct dquot_operations       *dq_op;
```

```
        struct quotactl_ops    *s_qcop;
        struct export_operations *s_export_op;
        unsigned long          s_flags;
        unsigned long          s_magic;
        struct dentry          *s_root;
        struct rw_semaphore    s_umount;
        struct mutex           s_lock;
        int                    s_count;
        int                    s_syncing;
        int                    s_need_sync_fs;
        atomic_t               s_active;
#ifdef CONFIG_SECURITY
        void                  *s_security;
#endif
        struct xattr_handler   **s_xattr;

        struct list_head       s_inodes;
        struct list_head       s_dirty;
        struct list_head       s_io;
        struct hlist_head      s_anon;
        struct list_head       s_files;

        struct block_device    *s_bdev;
        struct list_head       s_instances;
        struct quota_info      s_dquot;

        int                    s_frozen;
        wait_queue_head_t      s_wait_unfrozen;

        char s_id[32];

        void                   *s_fs_info;
        /* Filesystem private info */
        /*
         * The next field is for VFS *only*.
         * No filesystems have any business
         * even looking at it. You had been warned.
         */
```

```
        struct mutex s_vfs_rename_mutex;      /* Kludge */

        /* Granularity of c/m/atime in ns.
           Cannot be worse than a second */
        u32               s_time_gran;
};
```

표 2.1 VFS의 수퍼블록 구조체

표2.1에서와 같이 파일시스템의 블록 사이즈 등 파일시스템 관리에 필요한 정보들을 가지고 있고 특정 파일시스템의 수퍼블록 오퍼레이션을 함수 포인터 테이블 형태로 유지한다. 특정 파일시스템이 자체의 슈퍼블록, 혹은 그에 상응하는 정보를 VFS에 수퍼블록의 내용을 채우면 VFS는 그 파일시스템에 대한 정보를 알게 되고 관리를 할 수 있게 된다. 표2.1에서 VFS의 많은 필드들을 가지고 또 오퍼레이션을 가지는데 여기서 상당 부분은 VFS에서 자체 처리를 하여 파일시스템 구현 시 대신 처리하는 역할을 하므로 개발자는 필요한 정보만 VFS에 변환하여(translation) VFS의 객체에 유지하면 되므로 많은 부분들을 구현하지 않아도 된다. 이 원리는 수퍼블록 객체에만 적용되는 것이 아니라 VFS의 모든 객체에 해당한다. 객체들은 객체 지향 방식이므로 같은 형태를 유지하고 있다.

나. Inode Object

VFS의 공통 파일 모델은 모든 장치, 파일, 디렉터리 등을 파일로서 관리하고 처리한다. 아이노드 객체는 일반 파일, 디렉터리, FIFO 등에 대한 VFS 객체이다. 실제 파일과 1:1 대응 관계를 가지며 디렉터리 엔트리와는 1:N의 관계를 가진다. User의 관점에서 보면 파일은 하나의 객체로 보이지만 파일시스템 내에서는 메타 데이터와 데이터 블록으로 이루어져 있다. 아이노드는 실제 파일에 대한 메타 데이터 즉, 파일을 식별할 수 있는 정보들을 가진다. 예를 들면 아이노드 번호, 파일 크기, 생성 시간 등. 그러므로 아이노드의 개수는 실제 파일의 개수와 일치한다. 단 유저레벨에서는

하드링크의 경우 파일이 2개인 것처럼 보일 수는 있으나 이것은 같은 아이노드를 가르키고 있다. 즉 디렉터리 엔트리만 추가되고 링크 카운트만 증가할 뿐 아이노드가 새로 생성되지는 않는다. 아이노드 역시 아이노드를 관리하기 위한 속성들과 오퍼레이션을 가진다.

```
struct inode {
        struct hlist_node     i_hash;
        struct list_head      i_list;
        struct list_head      i_sb_list;
        struct list_head      i_dentry;
        unsigned long         i_ino;
        atomic_t              i_count;
        unsigned int          i_nlink;
        uid_t                 i_uid;
        gid_t                 i_gid;
        dev_t                 i_rdev;
        unsigned long         i_version;
        loff_t                i_size;
#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t            i_size_seqcount;
#endif
        struct timespec               i_atime;
        struct timespec               i_mtime;
        struct timespec               i_ctime;
        unsigned int          i_blkbits;
        blkcnt_t              i_blocks;
        unsigned short        i_bytes;
        umode_t               i_mode;
        spinlock_t            i_lock;
        /* i_blocks, i_bytes, maybe i_size */
        struct mutex          i_mutex;
        struct rw_semaphore   i_alloc_sem;
        struct inode_operations       *i_op;
        const struct file_operations  *i_fop;
/* former ->i_op->default_file_ops */
        struct super_block    *i_sb;
```

```
        struct file_lock       *i_flock;
        struct address_space   *i_mapping;
        struct address_space   i_data;
#ifdef CONFIG_QUOTA
        struct dquot           *i_dquot[MAXQUOTAS];
#endif
        struct list_head       i_devices;
        union {
                struct pipe_inode_info *i_pipe;
                struct block_device    *i_bdev;
                struct cdev            *i_cdev;
        };
        int                    i_cindex;
        __u32                  i_generation;
#ifdef CONFIG_DNOTIFY
        unsigned long          i_dnotify_mask;
/* Directory notify events */
        struct dnotify_struct  *i_dnotify;
       /* for directory notifications */
#endif



        unsigned long          i_state;
        unsigned long          dirtied_when;
       /* jiffies of first dirtying */

        unsigned int           i_flags;

        atomic_t               i_writecount;
#ifdef CONFIG_SECURITY
        void                   *i_security;
#endif
        void                   *i_private;
      /* fs or device private pointer */
};
```

표 2.2 VFS의 아이노드 구조체

아이노드는 검색 속도를 향상시키기 위하여 해시와 캐시로 관리된다. 파일시스템에서는 아이노드 혹은 이에 상응하는 정보를 VFS에 전달하여야

한다. 특히, 마운트 시에는 파일시스템의 루트 아이노드가 VFS에 전달되어지는 데 이 시점에서 아이노드 오퍼레이션을 포함한 파일시스템에 특정적인 오퍼레이션과 정보들이 활성화 되게 된다. 따라서 아이노드가 등록되는 시점부터 그 아이노드 종류에 일치하는 함수들을 호출 할 수 있게 된다.

다. File Object

파일 객체는 열린 파일과 그 열린 파일을 소유한 프로세스와의 관계를 나타내는 객체이다. 실제로 장치(예. 디스크)에 존재하지 않고 메모리상에서만 존재하는 동적인 객체이다. 프로세스가 소유하는 동안 존재한다.

```
struct file {

        union {
                struct list_head       fu_list;
                struct rcu_head        fu_rcuhead;
        } f_u;
        struct path            f_path;
#define f_dentry       f_path.dentry
#define f_vfsmnt       f_path.mnt
        const struct file_operations  *f_op;
        atomic_t               f_count;
        unsigned int           f_flags;
        mode_t                 f_mode;
        loff_t                 f_pos;
        struct fown_struct     f_owner;
        unsigned int           f_uid, f_gid;
        struct file_ra_state   f_ra;

        unsigned long          f_version;

        void                   *private_data;

        struct address_space   *f_mapping;
};
```

표 2.3 VFS의 파일 구조체

메모리 상에 존재하므로 VFS에서 제공하는 디폴트 오퍼레이션을 사용할 수도 있고 파일시스템의 디자인에 따라 별도로 구성할 수도 있다.

라. Dentry Object

디렉터리의 각 항목에 대한 정보를 나타내는 객체이다. Linux에서는 디렉터리 역시 파일로 처리되므로 dentry 정보는 디렉터리와 파일의 항목을 동일시하고 관리한다[4].

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;          /* protected by d_lock */
        spinlock_t d_lock;             /* per dentry lock */
        struct inode *d_inode;         /* Where the name belongs to -
                                          NULL is * negative */
        struct hlist_node d_hash;      /* lookup hash list */
        struct dentry *d_parent;       /* parent directory */
        struct qstr d_name;

        struct list_head d_lru;                /* LRU list */

        union {
                struct list_head d_child;/* child of parent list */
                struct rcu_head d_rcu;
        } d_u;
        struct list_head d_subdirs;    /* our children */
        struct list_head d_alias;      /* inode alias list */
        unsigned long d_time;          /* used by d_revalidate */
        struct dentry_operations *d_op;
        struct super_block *d_sb;   /* The root of the dentry tree */
        void *d_fsdata;                /* fs-specific data */

        int d_mounted;
        unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /*  small  names
*/
};
```

표 2.4 VFS의 dentry 구조체

디렉터리 엔트리 객체에서 가장 중요한 오퍼레이션은 디렉터리 항목을 빠르게 찾기 위한 Hash 함수라고 볼 수 있다. 각 파일시스템은 디렉터리 구조가 다르므로 이러한 오퍼레이션을 각자 구현하거나 성능에 문제가 되지 않는 다면 구현하지 않고 디폴트 오퍼레이션 사용이 가능하다. 디렉터리 엔트리 역시 캐시로 유지되어 한 번 읽은 엔트리는 빠르게 다시 접근할 수 있다. 디렉터리는 부모와 형제, 자식 노드들을 가지고 있어 트리 형태의 구조를 이룬다.

마. Filesystem Type과 Address Space Object

이 2개의 객체는 운영체제 혹은 파일시스템 관련 책에서 주로 논의되는 4대 객체에 속하지는 않지만 구현의 관점에서는 역시 중요한 역할을 한다.

파일시스템 타입은 VFS가 파일시스템의 종류를 식별하는 데 필요하다. 마운트 시에 등록이 되어 VFS가 파일시스템을 지원가능 하게 한다.

```
struct file_system_type {
        const char *name;
        int fs_flags;
        int (*get_sb) (struct file_system_type *, int,
                       const char *, void *, struct vfsmount *);
        void (*kill_sb) (struct super_block *);
        struct module *owner;
        struct file_system_type * next;
        struct list_head fs_supers;
        struct lock_class_key s_lock_key;
        struct lock_class_key s_umount_key;
};
```

표 2.5 VFS의 파일시스템 타입 구조체

어드레스 스페이스 객체는 page cache에서 page를 그룹화하고 관리하는 데 사용한다. 실제로 대부분의 I/O는 page cache를 이용하므로 디바이스

를 이용하는 파일시스템은 이 객체를 꼭 구현해야 한다. 이 객체는 스토리지와 어플리케이션간의 중재자 역할을 한다. 커널 2.6에서는 page 단위 연산만 사용하지만 2.4에서는 블록 단위 연산도 병행하여 사용하였다.

커널 2.6 이상의 버전에서는 페이지 캐시를 사용하므로 파일에 대한 I/O는 이 객체를 통하여 이루어진다.

```
struct address_space {
        struct inode            *host;
       /* owner: inode, block_device */
        struct radix_tree_root page_tree;
       /* radix tree of all pages */
        rwlock_t                tree_lock;
       /* and rwlock protecting it */
        unsigned int            i_mmap_writable;
       /* count VM_SHARED mappings */
        struct prio_tree_root  i_mmap;
       /* tree of private and shared mappings */
        struct list_head        i_mmap_nonlinear;
       /*list VM_NONLINEAR mappings */
        spinlock_t              i_mmap_lock;
       /* protect tree, count, list */
        unsigned int            truncate_count;
       /* Cover race condition with truncate */
        unsigned long           nrpages;
       /* number of total pages */
        pgoff_t                 writeback_index;
       /* writeback starts here */
        const struct address_space_operations *a_ops;/* methods */
        unsigned long           flags;  /* error bits/gfp mask */
        struct backing_dev_info *backing_dev_info;
       /* device readahead, etc */
        spinlock_t              private_lock;
       /* for use by the address_space */
        struct list_head        private_list;  /* ditto */
        struct address_space   *assoc_mapping;        /* ditto */
} __attribute__((aligned(sizeof(long))));
```

표 2.6 VFS의 어드레스 스페이스 구조체

2.2 기존 연구된 파일시스템

2.2.1 리눅스 공식 파일시스템

리눅스에서는 커널 2.6.20에서 60여개의 파일시스템을 지원한다. 이렇게 다양한 지원이 가능한 것은 VFS가 추상화 레이어로서 잘 정의 되어 있기 때문이다. 현재 리눅스의 표준 파일시스템은 ext3 파일시스템이다.

리눅스는 초기 개발 당시의 환경은 타넨바움 교수의 교육용 운영체제 minix였고 많은 부분을 비슷한 형태로 구현했다. 파일시스템 역시 초기에 미닉스 파일시스템을 사용하였다. 여기서 사용자들과 Linus는 2가지 심각한 문제를 발견하게 되는데 그 중 첫 번째는 블록의 주소지정이 16bit integer로 파일시스템의 크기가 64MB로 제한 되었고, 두 번째는 파일이름의 길이를 14글자로 제한되었다. 리눅스 파일시스템 개발자들은 이 문제들을 해결하기 위해 Extended File System을 개발했지만 비슷한 모습을 유지하였기 때문에 여전히 문제가 존재했고, 더 향상된 파일시스템이 필요했고 파일시스템의 추가가 쉬운 구조가 필요했다. 그래서 VFS를 개발하고 Extended2를 개발하기에 이른다. 리눅스 커뮤니티를 통해 테스트가 이루어지면서 1992년 이래로 점진적인 발전으로 이루면서 저널링을 지원하는 ext3에 구현과 안정화를 이루어 낸다.[3]

각종 vendor에서도 리눅스용 파일시스템 혹은 자사의 파일시스템을 리눅스에 포팅했다. IBM의 JFS와 CIFS, RedHat의 GFS2, Silicon Graphics의 XFS 등이 있고, 심지어는 타 운영체제의 파일시스템들이 포팅 되었다. 윈도우의 FAT, NTFS 등 다양한 환경의 파일시스템들이 리눅스에서 사용이 가능하다. 현재 리눅스 커널에 포함된 이 파일시스템들은 신뢰성과 안정성에 대해 증명이 되어 있고 여러 가지 용도에 응용이 가능하다. 대부분 기업 혹은 대형 커뮤니티 수준에서 수년간 유지 보수되고 관리되고 있

기 때문에 성능 또한 뛰어나다. 일부 포팅 예를 들면 2.4에서의 NTFS는 쓰기 연산에 그리 안정적이지 못했는데 현재 쓰기가 가능하며 안정성 면에서 크게 향상 되었다.[11]

또한 FUSE(filesystme for userspace)는 커널모듈과 응응프로로그램 라이브러리를 제공해 유저 레벨의 파일시스템의 제작이 가능하게 한다. 이것을 이용하면 간단한 코드만으로도 파일시스템을 만들 수 있다.[11]

리눅스의 파일시스템은 다양한 목적과 방법으로 견고한 모습으로 발전하고 있으며 그 응용 또한 다양하다. 타 시스템에서 개발 된 파일시스템을 VFS에 적용하여 동작하게 하는 부분도 잘 지원이 되어서 리눅스의 파일시스템의 종류는 갈수록 증가할 것이다.

2.2.2 국내 개발 동향

국내 리눅스 파일시스템 개발은 3가지 흐름이 있다. 첫째, 파일시스템 디자인 제안, 둘째 기존 파일시스템의 확장, 셋째 파일시스템의 구현이다[5][6][7][10].

디자인 제안이나 파일시스템의 확장에 관한 연구와는 달리 3번째인 파일시스템의 완전한 개발을 살펴보면 공통점을 가지고 있는 데 그것은 리눅스의 VFS와는 별도의 파일시스템으로서 구현이 되어 있다는 것이다. 물론, 각 연구들은 각 연구 목적에 부합하는 파일시스템을 개발하였지만 하나의 운영체제 안에서 같은 역할을 하는 2개의 서브시스템을 가지고 있는 형태가 되었다. VFS에 무관하게 동작하는 파일시스템은 그 구조에 따라 다르지만 개발이 훨씬 단순할 수는 있다. 하지만 이런 장점이 있는 반면 치명적인 단점이 있다. 먼저, 호환성이 부족하다. 표준 시스템 콜이 VFS에 요청하기 때문에 VFS에 무관하게 동작하는 파일시스템은 새로운 인터페이스를 응용 프로그램에 제공하여야 한다. 그리고 Linux 운영체제

는 이것이 파일시스템인지조차 인식할 수 없다. 또 자원을 효율적으로 사용할 수 없다. 파일시스템들이 공유할 수 있는 자원들을 2개의 서브시스템으로 나눠서 사용하여야 하므로 효율성이 떨어진다. 마지막으로 2개의 같은 시스템이 있으므로 이것에는 당연히 기능 중복의 문제가 있다. 같은 기능이 필요하더라도 재사용이 불가능하다.

이러한 문제점들은 VFS의 하위 모듈로서 작성하게 되면 이러한 문제는 일시에 해결할 수 있으며, 본 연구는 위와 같은 새로운 파일시스템들이 쉽게 이식이 되는 것 또한 목표로 하고 있다. 이식이 쉽다면 파일시스템 유지보수도 쉬워져 기술이전이 용이해져 연구가 사장될 위험성을 줄일 수 있다.

2.3 VFS 안에서 파일시스템 개발의 어려움

이전의 단락에서 VFS로 인한 파일시스템의 개발의 용이성을 설명하고 강조하였으나, 이번에는 그 어려움에 대해 역설하고자 한다. 개념적으로는 VFS와 특정 파일시스템의 분리가 잘 되어 있으나, 구현적인 관점으로는 굉장히 혼합되어 있는 상태이다. 그러므로 VFS의 소스코드를 면밀히 분석하지 않고서는 파일시스템 개발 및 포팅은 굉장히 어려운 작업이 된다. 이번 단락에서는 파일시스템 개발의 어려움의 요소들을 나열하고 이 연구를 통해 그러한 문제들을 어떻게 해결했는지를 기술한다.

2.3.1 문서의 부재

보통 리눅스에서의 개발을 할 때 많은 문서들을 인터넷 혹은 책 등 다양한 자원으로부터 많은 정보를 얻을 수 있다. 하지만 파일시스템에 관한 부분은 오랫동안 다양하게 개발이 되고 있음에도 불구하고 문서화가 부족

하다. 특히, VFS와 특정 파일시스템이 연결되고 상호 작용하는 부분에서는 설명이 되어있는 문서들이 전무하다. 유일한 문서라면 VFS와 각 파일시스템의 소스코드이다. 파일시스템을 개발하기 위해 필요한 관련 연구 자체가 힘들어지게 하는 요소이다.

이 문제에 대한 인식은 본 연구에서 밝혀진 것이기도 하지만 요즈음의 리눅스 커널 커뮤니티에서도 종종 거론되고 있는 문제이고 커져가는 커널 개발자간의 기술격차에도 심각히 우려하고 있는 실정이다. CIFS의 maintainer가 리눅스 심포지움 2006에서 아주 간단한 메모리에 상주하는 파일시스템 개발 튜토리얼을 발표한 사례가 있다. 하지만 이것이 커뮤니티 공식적인 움직임이 아니기 때문에 체계적인 문서화가 이루어지 않고 있고 여전히 파일시스템을 개발하기에 문서들은 매우 부족한 편이다.

파일시스템을 개발하기 위해 본 연구에서는 꼭 필요한 부분을 밝히고 그에 필요한 문서를 제공하려 한다. 문서의 부재에도 불구하고 열람할 수 있는 소스코드가 존재하므로 그것을 통해 새로운 문서화가 가능하고 새로운 파일시스템 템플릿을 구현하기에 시행착오가 있겠지만 초기의 어려움을 극복하면 파일시스템 코드에서 공통적인 부분을 추출할 수 있고 나아가 그 공통적인 부분 또한 개발자들에게 좋은 문서가 될 수 있다.

본 연구는 파일시스템 개발에 필요한 최소한의 문서, 꼭 필요한 문서를 제공하고 그 문서를 참고하여 간단한 파일시스템을 개발하여 실제 파일시스템의 템플릿을 제공한다.

2.3.2 VFS의 복잡성과 커널 개발의 어려움

현재 VFS의 소스코드는 약 55,000 줄 이상으로 이루어 졌고, 상대적으로 간단하다고 볼 수 있는 미닉스 파일시스템이 약 2,500 줄 정도 이다. 그래서 파일시스템을 이해하기 위해 읽어야 할 소스 코드는 적어도 57,500

줄 이상이다. 코드의 크기가 클 뿐만 아니라 VFS는 코드 자체가 C 언어로 객체 지향 언어 형식으로 작성으로 되어 있어 실제 C++/Java 처럼 객체 지향이 직관적이지 않으므로 코드의 모습이 상당히 복잡하면서 리눅스 커널의 다른 서브시스템과 상호 작용하는 부분이 많아 VFS만 보고 이해하기 어렵다. 또한 리눅스의 커널 코드는 버전 업에 따라 주석 없이 코드가 최적화를 위해 간략해 지는 경향이 있어 또 그 생략된 정보를 이해하여야 한다. 또한 관용적으로 사용되는 코드들 또한 논리적인 흐름으로 되어 있는 것이 아니기 때문에 이해하는 데 상당한 어려움이 있다. 본 연구에서는 minix fs와 ext2 fs의 비교 분석을 통해 범위를 파일시스템과 VFS와의 관계로 제한하여 그 관계를 명확히 밝혀내고 특정 파일시스템이 공통적으로 개발하여야 부분을 찾아 API 형식으로 문서화 한다.

다른 관점으로 커널 개발의 측면에서 커널 개발은 여러 가지 어려움을 포함한다. 사소한 포인터 실수는 전체 시스템의 정지를 초래한다. 시스템이 정지하므로 문제의 원인을 찾는 과정과 디버깅이 어려워진다. 커널에서는 응용프로그램에서 쓸 수 있는 데이터 타입의 종류에 제한이 있고, 스택의 크기에도 제한이 있다. 개발환경이 열악하여 생산성이 저하되므로 본 연구의 본질적인 목적은 아니지만 최적의 파일시스템 개발 환경을 제안하여 개발환경 때문에 소비되는 시간을 최대한 줄이고자 한다. 문서화, 템플릿 코드, 개발환경 이 3가지가 파일시스템 개발 방법 제공의 요소가 되겠다.

2.4 파일시스템 개발의 위한 템플릿의 필요

파일시스템은 그 디스크 레이아웃 구성과 메타 데이터의 활용에 따라 그 모습을 달리 한다. 하지만 VFS의 공통 인터페이스에 대한 부분은 호환성을 위해 꼭 구현해야 하는 부분이며 그 부분은 파일시스템에 따라 조

금씩 차이는 있지만 같은 원리가 적용되므로 이해하기 쉬운 템플릿을 제공한다면 파일시스템의 개발 시작점으로서 공통의 코드의 중복을 피하여 개발의 오버헤드를 줄일 수 있다.

2.4.1 공통 기본 코드

여기서 공통 기본 코드라고 하면 VFS와의 상호 관계 즉 VFS의 공통 인터페이스에 필요한 파일시스템 정보 추출과 오퍼레이션 등록 부분이다. 이러한 부분은 리눅스에서 VFS의 하위 모듈로 동작하고자 하는 모든 파일시스템이 필수적으로 구현해야 할 부분이다. 이런 부분들이 명시적으로 드러나 있지는 않지만 파일시스템 간의 비교를 통해 밝혀 낼 수 있다. 본 연구에서는 리눅스의 파일시스템의 시조인 미닉스 파일시스템을 기본으로 하여 리눅스 표준 파일시스템인 ext2,3 파일시스템과의 비교를 통해 최대한 공통의 부분을 찾아내어 파일시스템의 템플릿을 작성한다. (미닉스 파일시스템은 순수 미닉스 파일시스템이 아니라 리눅스에 포팅된 버전이다.)

공통의 코드라도 파일시스템은 각기 다른 디스크 레이아웃과 다른 메타데이터의 활용으로 완전한 동일한 코드 템플릿을 제공할 수는 없으나 그 공통 코드에서 수행하여야만 하는 작업들에 관한 정확한 기술을 할 수 있어 코드 자체가 문서가 될 수 있다. 공통 코드 외에도 고려해야 할 사항은 여전히 존재하며 파일시스템 개발의 어려움을 완전히 극복할 수는 없다. 하지만 공통의 코드를 구현한 부분들을 통해 VFS와 특정 파일시스템 과의 상호 작용을 이해한다면 파일시스템의 리눅스 호환이 어렵지 만은 않은 작업이 될 것을 기대한다. 본 연구에서는 파일시스템 개발 시 파일시스템 자체에 집중을 하고 VFS에 호환되는 부분에 필요한 노력을 최소화 하고자 한다.

2.4.2 개발을 위해 연구되어야 할 내용

여기서는 파일시스템을 개발하기 위해 알아야 할 기초적인 내용과 구현을 위해 알아야 할 내용을 기술한다.

기초적인 내용들은 다음과 같다.

- OS 일반지식
- 리눅스 커널의 기본적인 이해
- 리눅스 파일시스템의 기본 원리
- C code의 독해 능력과 작성 능력

위의 4가지가 준비되었다면 파일시스템의 구현이 가능하다 할 수 있겠다. 본 연구에서는 위의 내용들을 다루지는 않는다. OS 수업을 통해 많은 부분을 갖추게 되었다고 가정한다.

구현을 위해 필요한 부분은 다음과 같다.

- VFS 의 이해
- filesystem driver의 동작 방식 이해
- 간단한 특정 파일시스템의 이해
- 데이터 블록 할당 방식의 이해
- 리눅스 시스템 콜의 흐름의 이해
- 저장장치에 대한 이해

상당히 많은 지식이 필요하지만, 이 연구에서는 최소한의 지식으로 리눅스 파일시스템의 구현을 목표로 한다. 위의 내용들은 깊이가 깊어질수록 파일시스템에 대한 시야도 넓어지며 구현을 보다 쉽게 한다. 하지만 그 소요되는 시간이 상당하므로, 실용적인 관점으로서 본 논문을 통해서 파일시스템을 이해하고 구현할 수 있는 지름길을 제공하고자 한다.

제 3 장 연구 내용

3.1 VFS와 리눅스 파일시스템 분석

VFS와 특정 파일시스템에 대한 상호관계 공통의 인터페이스를 파악하기 위해 리눅스에서 비교적 간단한 구조를 가지고 있는 미닉스 파일시스템을 분석한다. 파일시스템은 단순하게 보면 실제 데이터와 그 데이터를 관리하기 위한 메타데이터들로 구성된다. 또 이런 메타 데이터와 데이터를 저장장치에 어떻게 구성할 것인가가 디스크의 레이아웃을 결정하게 된다. 미닉스 파일시스템은 이러한 구조가 초창기 유닉스 모델을 따르면서도 간단하게 구성되어 있어 파일시스템 연구로 좋은 시작점이 되기는 하나 역시 수년간 재작성 되어 코드가 간결해짐에 따라 이해가 어려운 점도 있다. 하지만 상대적으로 작은 크기의 미닉스 파일시스템을 통해 VFS와 파일시스템의 동작 방식을 이해한다.

3.1.1 미닉스 파일시스템의 구조

가. Disk Layout

디스크 레이아웃은 실제 저장장치에 파일시스템이 물리적으로 가질 형태이다. 파일시스템은 대부분 블록단위로 메타데이터를 관리하며 미닉스의 디스크 레이아웃은 그림3.1과 같다.

첫 번째 부트 블록은 말 그대로 부팅을 위한 공간이며 부팅디스크가 아니라면 필요없지만, 호환성을 위해 존재한다. 실제 구현에서는 구현하지 않아도 파일시스템의 동작과는 상관없지만 본 템플릿에서는 이해를 돕기 위해 부트 블록은 유지한다. .

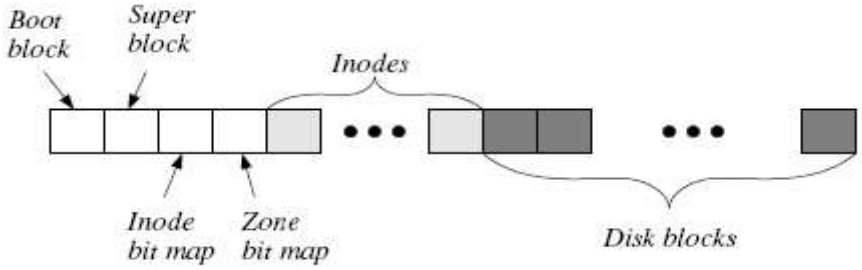

그림 3.1 미닉스 파일시스템 레이아웃

두 번째 수퍼블록은 미닉스 파일시스템의 정보를 가진다. 수퍼블록은 파일시스템을 컨트롤하며 나머지 메타데이터에 대한 관리도 하고 파일시스템의 현재 상태를 통해 파일시스템을 검사할 것인가에 관한 정보도 제공한다. 그림3.2의 상단 부분은 디스크와 메모리에 동시에 저장 되어야 하는 정보들을 나타내고 하단 부분은 메모리상에서만 존재하는 정보를 나타낸다. 메모리상에서만 존재한다는 의미는 VFS에서 필요한 정보를 의미한다. 메모리와 디스크에 동시에 적용되는 정보는 주기적으로 동기화가 이루어진다.

두 번째와 세 번째 아이노드와 존에 대한 비트맵이다. 먼저 비트맵은 그 요소가 사용 중 인지를 bit 값으로 즉 0 or 1로 나타낸다. 비트 연산을 통해 효율적으로 사용 가능한 요소를 빠르게 찾을 수 있고, 사용 유무를 쉽게 관리할 수 있다. 이 방식은 많은 파일시스템에서 채택하고 있는 방식이다. 존은 미닉스에서 데이터 블록을 가리키는 이름이다. 즉 아이노드와 데이터블록은 bitmap으로 관리한다.

네 번째는 Inode들의 모임인 아이노드 테이블이다. 아이노드의 개수는 전체 스토리지의 크기에 따라 개수가 증가할 수 있고, 이 아이노드들은 아

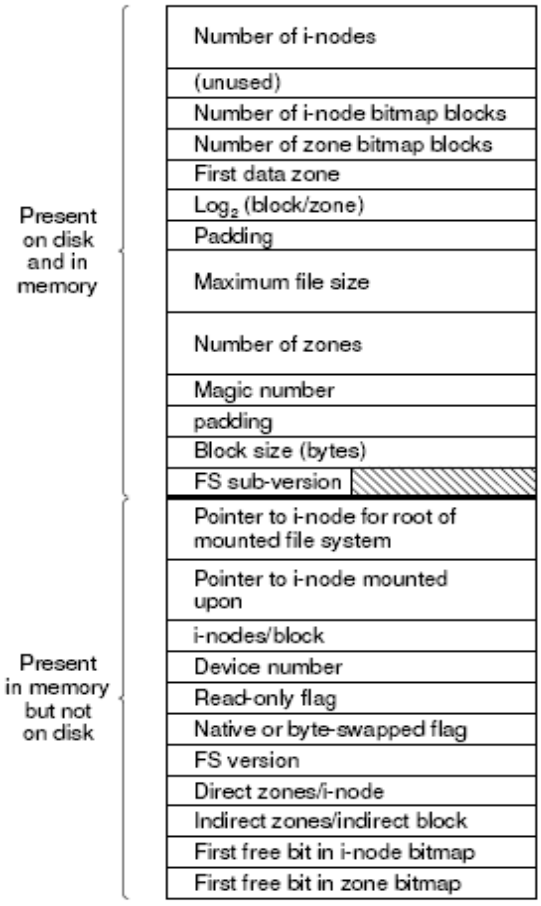

그림 3.2 미닉스 아이노드 구조

이노드 맵의 순서와 일치한다. 따라서 아이노드들은 객체 정보에 아이노드 번호를 가지지 않는다. 아이노드의 위치(순서)가 번호와 일치하기 때문에 아이노드 번호가 별도로 필요하지 않기 때문이다.

아이노드는 파일자체에 관한 메타데이터를 저장하고 있으며 실제 데이터블럭에 대한 포인터(Zone N)를 가진다. 미닉스 아이노드는 64byte이고 1K 블록 단위라면 블록당 128개의 아이노드를 갖게 된다.

아이노드 테이블 이후에 블록들은 실제 데이터가 저장되어 있는 블록이다. 파일에 대해 불연속적으로 존재하지만 유저레벨에서는 순차적인 스트림으로 인식한다. 여기서 파일의 데이터 블록들의 배치가 실제로 연속적이

라면 하드디시크의 동작 방식에 따라 파일 접근 시 읽기 속도가 빨라진다. 데이터 블록을 할당할 시 읽기속도를 고려한다면 데이터 블록을 어떻게 할당할 것인가도 파일시스템 디자인 시 고려 대상이 된다.

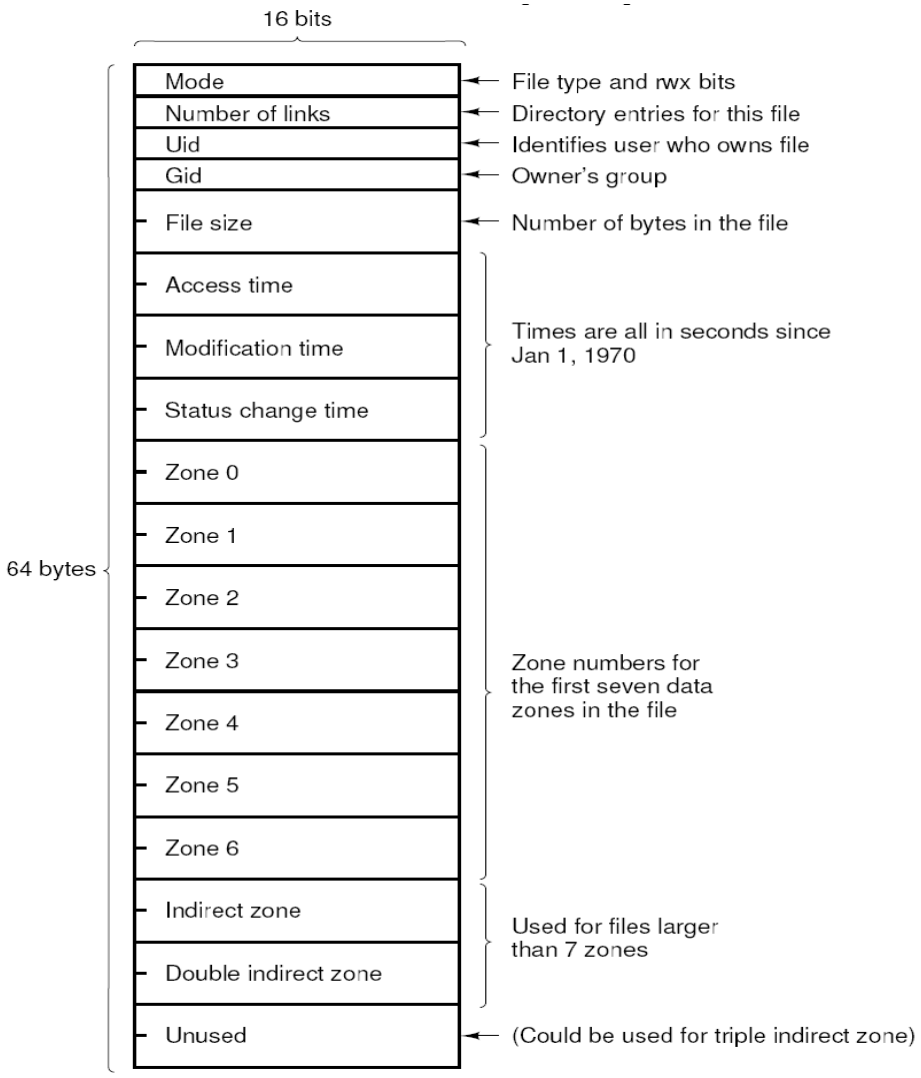

그림 3.3 미닉스 디스크 아이노드의 구조

나. Source Code Tree

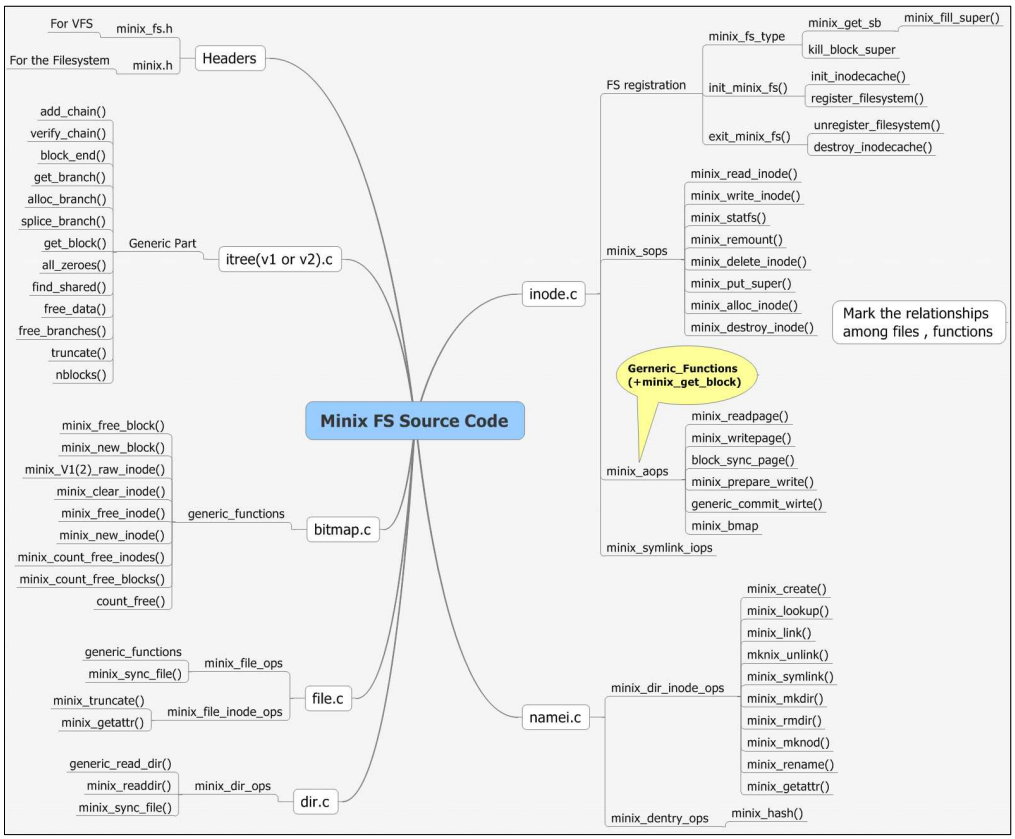

그림 3.4 미닉스 파일시스템 소스 트리

미닉스에서는 16byte, 32byte에 따라 미닉스 파일시스템 버전 1, 2로 사용할 수 있으며 버전은 파일시스템 생성시 결정된다.

inode.c 에서는 파일시스템 등록 오퍼레이션, 수퍼블럭 오퍼레이션, 어드레스 스페이스 오퍼레이션이 구현되어있다. namei.c 에서는 디렉터리 아이노드 오퍼레이션과 디렉터리 엔트리 오퍼레이션이 구현 되어 있다. dir.c에서는 실제 스토리지에서 디렉터리 엔트리에 관한 관리에 관한 부분이 구현되어 있다. file.c는 파일에 관한 오퍼레이션이 구현되어 있고 bitmap.c에서는 아이노드와 존에 관한 비트맵 오퍼레이션이 구현 되어 있다. itree.c

에서는 버전에 따라 블록 포인터에 관한 오퍼레이션이 구현 되어 있다.

3.1.2 VFS와 미닉스 파일시스템

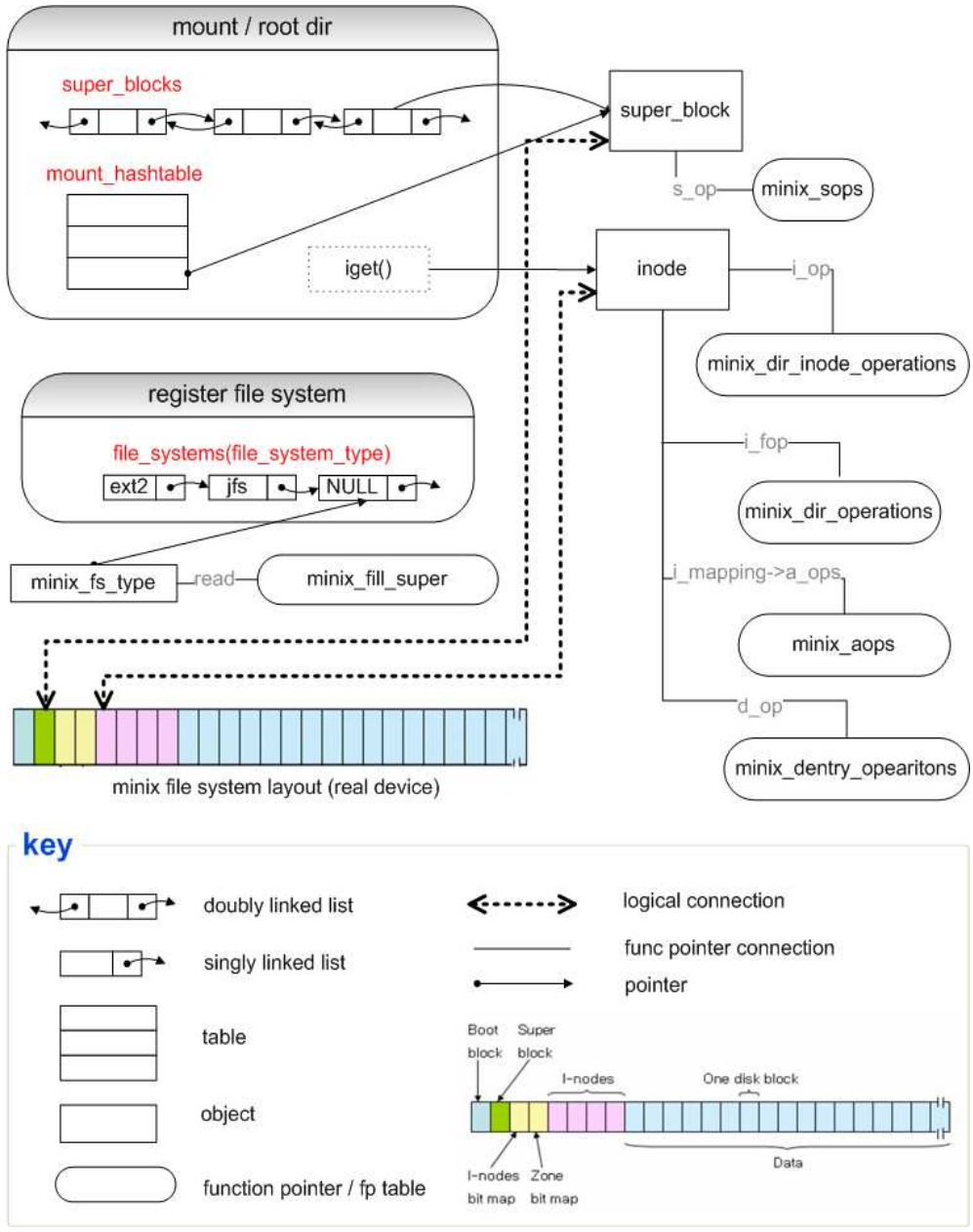

그림 3.5 VFS와 minix 파일시스템의 마운트시 스냅샷

이 그림은 마운트시 파일시스템이 등록되고 난 직후의 상태를 설명한다. 마운트시 미닉스 파일시스템은 자신의 파일시스템을 등록한다. 등록시에 파일시스템의 get_sb() 함수는 minix_fill_super()함수를 콜백 함수로 등록하고 minix_fill_super() 함수를 통해 실제 디바이스에서 수퍼블록을 읽어 VFS의 수퍼블록을 채운다. 이 수퍼블록은 VFS에서 연결 리스트로 관리하고 이 시점에서 미닉스 파일시스템은 iget()함수를 호출하여 자신의 루트 디렉터리를 시스템의 루트 파일시스템의 마운트시 정한 디렉터리에 마운트 한다. iget()함수는 minix_read_inode()함수를 호출하여 미닉스의 특정 오퍼레이션을 등록한다. 이렇게 함으로써 미닉스 파일시스템의 루트 디렉터리는 접근이 가능해진다. 위의 그림에서 보듯이 각각의 오퍼레이션은 아이노드의 종류에 따라서 다른 게 적용되며 처음 마운트 시와 동일하게 아이노드는 필요에 따라 메모리에 유지되게 된다. 실제 파일시스템에서는 등록된 오퍼레이션을 구현하고 객체의 정보를 읽고, 갱신함으로서 동작하게 된다. 위의 그림과 같은 연결은 다음의 코드로 인해 만들어진다.

```
static int minix_fill_super(struct super_block *s, void *data, int
silent)
{
        - 중략 -
        s->s_op = &minix_sops; // 수퍼블록 오퍼레이션 연결

        root_inode = iget(s, MINIX_ROOT_INO);
        // iget함수는 minix_read_inode()를 호출하고
        // minix_read_inode() 각 아이노드에 맞는 오퍼레이션을 할당한다.

        s->s_root = d_alloc_root(root_inode);
        // root dentry를 할당한다.

       - 중략 -
}
```

표 3.1 미닉스 파일시스템 fill_super()의 축약된 소스 코드

3.2 파일시스템 템플릿 디자인

3.2.1 파일시스템 템플릿의 요구사항

본 파일시스템은 특정 파일시스템의 개발 시작점이 될 수 있는 프레임웍을 제공한다. 따라서 디자인 목표는 현존하는 파일시스템들의 특성 중에서 공통적인 부분들을 가능한 표현하고 표준 파일시스템 요구사항(basic functionality)을 구현하는 것이다. 본 파일시스템은 다음의 3 가지를 기본 원칙으로 설계한다.

- 가능한 단순한 구조 유지

파일시스템이 가지고 있어야할 최소한의 메타데이터를 표현하는 범위 내에서 디스크 구성을 단순하게 하여 직관적 모습을 유지한다.

- UNIX의 파일 모델을 준수한다.

Linux VFS는 전통적인 Unix Common File Model을 그대로 따른다. 그러므로 디스크 구성 역시 가능한 동일하게 하여 설계와 구현의 외관적 차이를 줄여 개발을 용이하게 한다.

- 확장성을 고려한다.

본 파일시스템은 그 자체가 독립적인 파일시스템의 역할을 수행할 수 있고 본 시스템을 기본으로 하여 확장하는 것 또한 중요한 목표이므로 디스크 구성에 유연성을 고려해 파일시스템 복잡도를 낮춘다.

3.2.2 디스크 레이아웃

디스크의 구성은 아래 그림3.6에서 보여진다.

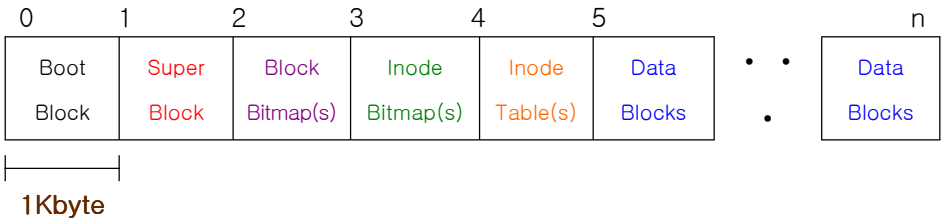

그림 3.6 파일시스템 템플릿 디스크 레이아웃

※ 주의 : 부트 블록은 첫 번째(0)에 위치하며 부팅 디스크가 아닐 경우라도 타 파일시스템들과의 호환성을 위하여 존재한다. 블록 2-4들은 장치(예: 하드디스크)의 용량에 따라 그 수가 가변적이므로 그림과는 달리 여러 블록들을 차지 할 수 있다. 하지만 실험을 위해 그림과 동일하게 1개 블록을 차지한다.

3.2.3 Data Structure

가. Superblock

```
struct sfs_super_block{
        u32 s_inodes_count;             // 아이노드 수
        u32 s_blocks_count;             // 데이터 블록 수
        u32 s_free_inodes_count;        // 가용한 아이노드 수
        u32 s_free_blocks_count;        // 가용한 데이터 블록 수
        u16 s_inodemap_blocks;          // 아이노드 맵 수
        u16 s_blockmap_blocks;          // 데이터 블록 맵 수
        u16 s_first_data_block;         // 첫 번째 데이터 블록
        u32 s_log_block_size;           // 블록 사이즈
        u32 s_mtime;
        u32 s_wtime;
        u32 s_max_size;                 // 파일시스템 최대 크기
        u16 s_magic;                    // 매직 시그니쳐
        u16 s_state;                    // 파일시스템 상태
};
```

표 3.2 파일시스템 템플릿의 수퍼블록

s_inodes_count

32bit 총 아이노드 개수(unused and free)

s_blocks_count

32bit 총 데이터 블록의 개수(unused and free)

s_free_inodes_count

32bit 총 사용 가능한(free) 아이노드 개수

s_free_blocks_count

32bit 총 사용 가능한(free) 블록의 개수

s_inodemap_count

16bit 생성된 inode map 이 차지하는 블록의 개수

s_blockmap_count

16bit 생성된 block map 이 차지하는 블록의 개수

s_first_data_block

16bit superblock을 포함하는 블록 번호, 보통 1

(superblock은 일반적으로 디스크의 3번째 섹터 즉 1024byte 번째에 위치한다.)

s_log_block_size

32bit 블록사이즈를 계산하는 데 사용

block size = 1024 << s_log_block_size

s_max_size

32bit filesystem이 가질 수 있는 최대 크기

(저장 장치(e.g disk)의 크기에 따라 달라짐,)

s_magic

16bit 파일시스템을 식별하는 번호

오래된 파일시스템과의 호환성을 위해 존재, 고정된 값을 가지며 중요하지 않은 값. 실제로 사용하지 않는다.

s_state

16bit 파일시스템의 상태를 나타냄

SFS_VALID_FS 정상

SFS_ERROR_FS 정상적으로 언마운트 되지 않음

나. Block Bitmap

블록 비트맵의 각 비트(bit)는 블록의 현재 사용 유무를 나타낸다. 1은 사용 중임(in-use)를 나타내고 0은 사용가능(미사용, free, unused)를 나타내며, 가용 블록(free block)을 빠르고 쉽게 찾는 기능을 한다. 이 비트맵은 디스크의 크기에 따라 그 크기가 비례한다.

다. Inode Bitmap

block bitmap 와 동일한 방식으로 사용하며 유일하게 다른 점은 아이노드의 사용 유무를 나타내는 것이다.

라. Inode Table

아이노드들의 모음이며, 각 요소는 실제 파일의 정보(파일의 크기, 위치, 액세스 권한...)를 담고 있다. 파일에 관한 가장 중요한 메타데이터이다. 아이노드에서는 파일의 이름은 더 이상 의미를 갖지 않고 아이노드 넘버로 식별된다. 이 객체에서 아이노드의 번호는 존재하지 않는다. 실제 아이노드 번호는 그 배열 순서와 일치하기 때문에 그 필드가 존재하지 않더라도 번호를 알 수 있기 때문이다.

```
struct sfs_inode {
        u16 i_mode;                     // 파일 접근 모드
        u16 i_link_count;               // 하드 링크 수
        u16 i_uid;                      // User ID
        u16 i_gid;                      // Group ID
        u32 i_size;                     // 파일 크기
        u32 i_atime;                    // 접근 시간
        u32 i_mtime;                    // 최근 수정한 시간
        u32 i_ctime;                    // 파일 생성 시간
        u32 i_data[10];                 // 블록 포인터 배열
};
```

표 3.3 파일시스템 템플릿의 아이노드

i_mode

16bit, 파일의 포맷과 접근 권한을 나타낸다.(conjunction)

i_link_count

16bit, 아이노드가 몇 번 링크되어 있는 지를 나타낸다.

i_uid

16bit 파일의 유저 아이디

i_gid

16bit 파일의 그룹 아이디

i_size

32bit 파일의 크기(byte)

i_atime

32bit 가장 최근 접근한 시간(EPOCH)

i_mtime

32bit 가장 최근 수정한 시간(EPOCH)

i_ctime

32bit 생성된 시간(EPOCH)

i_data[10]

32bit 블록 번호를 가지는 배열, 각각의 요소는 특정 파일의 블록 하나(direct) 위치를 나타낸다. indirect 블록의 경우 블록 번호들이 저장되어 있는 블록을 가르킨다. bi-indirect의 경우는 indirect가 모여있는 블록을 가르킨다.

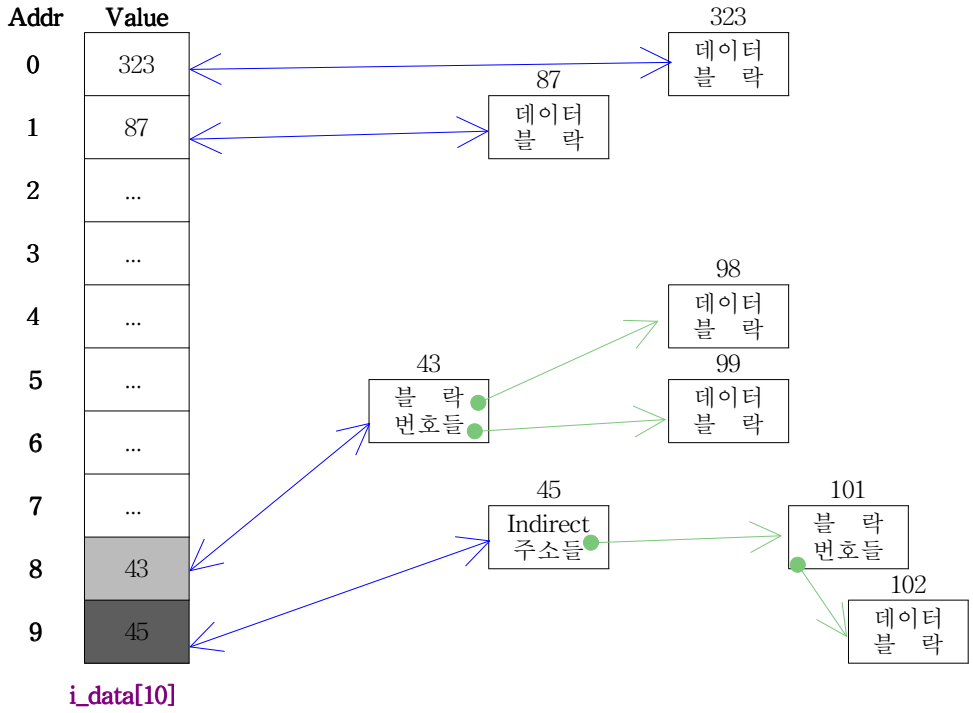

그림 3.7 블록 종류와 그 연결 방식

- 0-7(흰색 배경)의 블록들은 direct block이며 이것은 실제 데이터가 위치하는 디스크 상의 블록 번호를 가지고 있다. 따라서 이 블록들이 가지고 있는 번호를 통해 바로 실제 데이터 접근이 가능하다. 다이렉트 블록의 데이터 크기 제한으로
- 8 (옅은 회색) 블록으로 더 큰 크기를 표현한다. indirect block 이라 부르며 이것은 데이터 블록 1개를 가리키며 이 블록에는 다이렉트 블록에서 처럼 실제 데이터가 위치하는 블록 번호들을 가지고 있다. 인디렉트 블록으로도 모자르다면
- 9 (진한 회색) 블록을 사용한다. 이것은 bi-indirect block(or double) 이것도 인디렉트 블록과 유사한 원리로 하나의 데이터 블록을 표현하지만 이 안에는 인디렉트 블록들의 주소가 저장되어 있다. ※ 주의 : 위의 블록 번호들은 임의로 지정한 것이며 한 아이노드(파일)에 관련한 블록들은 이

상적인 경우 순차적이지만 보통 흩어져 있다. 구현된 파일시스템은 direct 블록만 다룬다. 그러므로 파일의 크기에 제한이 있다(1KB*10)

마. Data Blocks

유저 레벨의 파일의 내용을 저장하고 있는 블록들이며 하나의 파일은 그 크기에 비례하여 여러 개의 블록들을 포함할 수 있다. 블록단위가 고정적(1kbyte)이기 때문에 블록의 내부 단편화가 생기지만 스토리지 디바이스들의 크기가 현대에는 내부 단편화를 무시해도 될 만큼 방대하기 때문에 문제가 되지 않는다. 블록의 크기는 VFS에서 PAGE_SIZE(4KB)로 제한하기 때문에 512B-4096B에서 512의 배수로 지정되어야 한다. 이 사이즈보다 큰 블록을 사용하려면 가상 블록 개념으로 여러 개의 블록을 묶는 방식으로 해야 할 것이다.

바. Directory Structure

디렉터리는 UNIX 표준에 따라 파일과 동일하게 다루어지고 저장된다. 루트 디렉터리는 파일 시스템 생성 시 생성된다. VFS 관례에 따라 루트 디렉터리의 아이노드 번호는 항상 1로 한다.

ext2에서는 이 객체에 레코드 길이 필드를 추가하여 가변적인 길이로 사용 가능하게 하나 연산이 많아져 복잡하게 된다. 본 파일시스템에서는 복잡도를 낮추기 위해 고정된 레코드 길이를 이용한다.(lookup, insert algorithm)

```
struct sfs_dir_entry {
       u32 inode;                 // 아이노드 번호
       u8  name_len;              // 디렉터리 이름 길이
       char name[SFS_NAME_LEN];   // 파일 네임 255 자 이내
};
```

표 3.4 디렉터리 엔트리 객체

inode

32bit 아이노드 번호

name_len

8bit 엔트리의 이름 길이를 나타낸다. 파일 네임 비교시 이용된다.

name

파일의 실제 이름이 저장된다.

3.2.4 Kernel APIs

가. VFS 객체와 SFS 의 객체 비교

다음의 내용은 VFS에서 제공하는 파일시스템 인터페이스를 커널 내부 문서에서[4] 규정하는 VFS 메쏘드들의 함수 테이블 목록이다. 모든 함수를 구현하는 것이 아니라 필요한 부분만 구현하면 된다. 실제 구현을 함수와 VFS객체와의 매핑을 표를 통해 표현하였다.

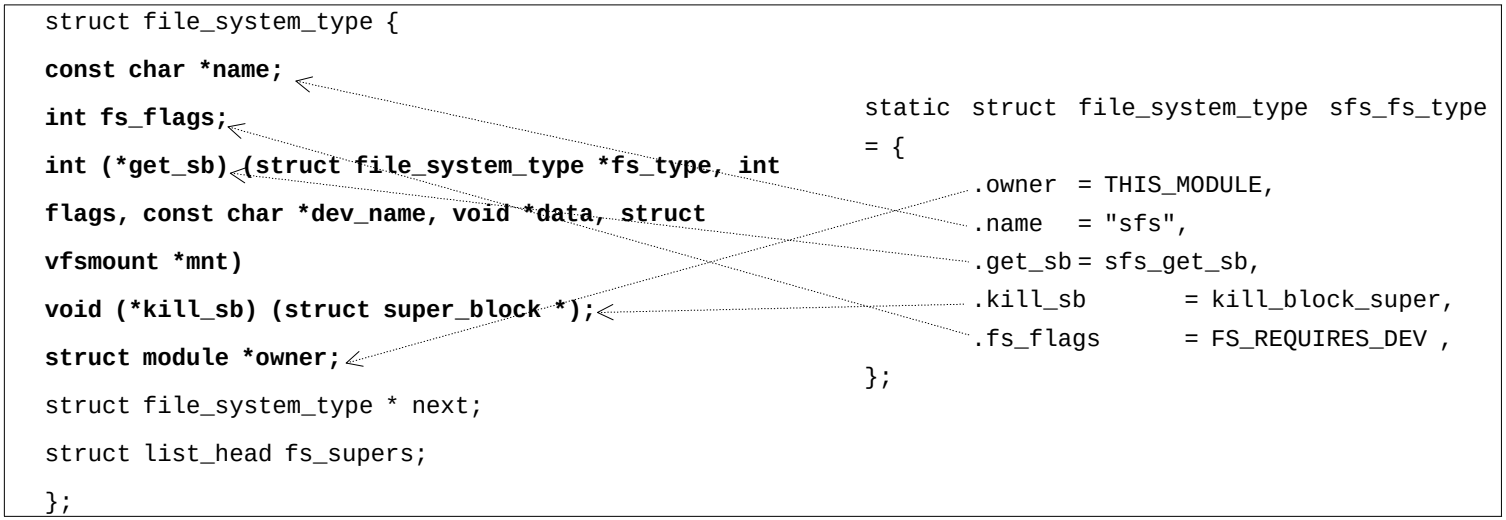

표 3.5 VFS 파일 시스템 타입와 파일시스템 템플릿의 파일 시스템 객체 비교

- Filesysem Type의 역할

이 객체는 파일시스템의 등록, 해제와 마운팅, 언마운팅에서 중요한 역할을 한다. 등록 시에 한 번만 초기화되고 해제될 때까지 커널 메모리에 유지된다. 파일시스템의 등록과 해제를 위해 아래와 같은 함수를 호출한다.

```
#include <linux/fs.h>
extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
```

struct file_system_type은 실제 파일시스템의 종류를 표현하며 이 구조체를 인자로 파일시스템을 VFS 파일

시스템 리스트에 등록하게 된다. 마운트 시에 VFS는 등록된 파일시스템 구조체에서 get_sb() 함수를 호출하여 특정 파일시스템의 수퍼블록의 정보를 가져오게 된다. 그리고 특정 파일시스템의 루트 아이노드를 등록하여 루트 디렉터리를 마운트 포인트에 새로운 디렉터리 엔트리로 갱신한다. 이 과정이 모두 이루어지면 시스템에서는 특정 파일시스템의 디렉터리 트리를 볼 수 있게 된다. 이 객체는 VFS의 추상 객체는 아니고, 특정 파일시스템을 식별하기 위한 별도의 구조체이고 파일시스템의 등록과 해제에만 관여하고 I/O 시에는 아무동작도 하지 않는다.

※ 주의 : 수퍼블록의 내용을 읽어오는 오퍼레이션이라 수퍼블록 객체에 포함 되어 있을 거라고 생각 할 수 있지만 VFS는 수퍼블록 객체를 생성한 이후에 수퍼블록 오퍼레이션을 쓸 수 있으므로 파일시스템 등록시에 수퍼블록을 먼저 읽고 그 이후에 수퍼블록 오퍼레이션 호출이 가능하므로 이 객체에 get_sb()함수가 포함되어 있다.

필 드	설명
name	파일시스템 타입의 이름 예 ext2, iso9660, msdos, etc
fs_flags	파일시스템 플레그 FS_REQUIRES_DEV, 장치를 사용하는 파일시스템 FS_NO_DCACHE, 디렉토리 엔트리를 캐쉬하지 않음
get_sb	파일시스템이 마운트 시에 호출 되는 함수 예) $ mount -t filesystem_type /mount/sample_filsystem 수퍼 블록 구조체는 VFS에 부분적으로 초기화 하고 그 나머지는 get_sb()에 의 해 초기화 되어야 한다 get_sb() 함수는 수퍼블록에 명시된 블록 디바이스가 그 메쏘드가 지원하는 타임의 파일시스템을 포함하는 지를 결정해야만 한다. get_sb()가 채우는 가장 중요한 멤버는 s_op 파일드 이다. 이것은 다음 단계의 파일시스템 구현을 표현 하는 “struct super_operations” 포인터이다. 일반적으로 파읽시스템 은 generic get_sb() 구현을 사용하고 대신에 fill_super()함수를 사용한다. 다음과 같은 제네릭 메쏘드들이 있다. get_sb_bdev: 블록 디바이스에 존재하는 파일시스템 마운트 get_sb_nodev: 디바이스를 사용하지 않는 파일시스템 마운트 get_sb_single: 모든 마운트 시 그 개체를 공유하는 파일시스템 마운트 Arguments struct file_system_type *fs_type : 파일시스템 타입 int flags: mount flags const char *dev_name: 마운트 할 디바이스 이름 예) /dev/hda1 void *data: 특정 파일시스템에서 사용할 옵션 마운트 시에 사용할 수 있다. (ASCII string) int silent: whether or not to be silent on error 에러 메시지 출력 여부
kill_sb	언마운트시 호출 되는 함수
owner	VFS가 내부적으로 사용하는 변수로 대부분의 경우 THIS_MODULE로 초기화 하여야 한다.
next	VFS의 내부 사용 변수로 NULL 초기화 하여야 한다

표 3.6 파일시스템 타입 객체의 멤버

- Superblock Operation (struct super_operations)

```
struct super_operations {
struct inode *(*alloc_inode)(struct super_block *sb);
void (*destroy_inode)(struct inode *);
void (*read_inode) (struct inode *);
void (*dirty_inode) (struct inode *);
int (*write_inode) (struct inode *, int);
void (*put_inode) (struct inode *);
void (*drop_inode) (struct inode *);
void (*delete_inode) (struct inode *);
void (*put_super) (struct super_block *);
void (*write_super) (struct super_block *);
int (*sync_fs)(struct super_block *sb, int wait);
void (*write_super_lockfs) (struct super_block *);
void (*unlockfs) (struct super_block *);
int (*statfs) (struct dentry *, struct kstatfs *);
int (*remount_fs) (struct super_block *, int *, char *);
void (*clear_inode) (struct inode *);
void (*umount_begin) (struct super_block *);
void (*sync_inodes) (struct super_block *sb, struct
writeback_control *wbc);
int (*show_options)(struct seq_file *, struct vfsmount *);
ssize_t (*quota_read)(struct super_block *, int, char *,
size_t, loff_t);
ssize_t (*quota_write)(struct super_block *, int, const
char *, size_t, loff_t);
};
```

```
static struct super_operations sfs_sops = {
.alloc_inode   = sfs_alloc_inode,
.destroy_inode = sfs_destroy_inode,
.read_inode    = sfs_read_inode,
.write_inode   = sfs_write_inode,
.delete_inode  = sfs_delete_inode,
.put_super     = sfs_put_super,
.statfs        = sfs_statfs,
.remount_fs    = sfs_remount,
};
```

표 3.7 VFS 수퍼블록 오퍼레이션과 파일시스템 템플릿 수퍼블록 오퍼레이션 비교

- 수퍼블록 오퍼레이션 별 역할

alloc_inode: 이 함수는 inode_alloc() 에 의해 아이노드를 위한 메모리 할당과 그것의 초기화를 하기 위해 호출 된다. 정의 되어 있지 않으면 단순한 아이노드가 할당된다.

destroy_inode: 이 함수는 아이노드를 위해 할당된 자원들을 해제하기 위해 호출 되고 alloc_inode()가 정의 되었을 때만 필요하고 그 반대의 작업을 한다.

read_inode: 특정 아이노드를 마운트된 파일시스템에서 읽기 위해 호출되고 , 아이노드 넘버 멤버는 읽을 아이노드를 가르키기 위해 VFS에의 해 초기화 된다. 다른 멤버들은 이 함수에 의해 초기화 된다. iget()함수를 호출 할 경우 이 함수가 실행된다. read_inode() i_op 필드를 채울 책임이 있다 . 이것은 struct inode_operations(각 아이노드에 수행될 수 있는)에 대한 포인터이다.

dirty_inode: 아이노드를 더티 상태로 만든다.

write_inode: VFS가 아이노드를 디스크 에 write 할 때 호출되면 두 번째 파라 미터는 그 write 가 동기화 되는 지를 가르킨다. 모든 파일시스템들이 이 플레그를 체크 하지는 않는다.

put_inode: VFS 아이노드가 아이노드 캐쉬로부터 지워 질때 호출된다.

drop_inode: 아이노드에 대한 마지막 접근이 없어질 때 아이노드 락 스픈락과 함께 호출된다 . 일반적이 유닉스 파일시스템 의미의 파일시스템을 위해 NULL이 되거나 아이노드를 캐쉬하지 않는 파일시스템을 위해 generic_delete_inode()를 사용한다. generic_delete)inode() 의 행동은 put_inode()에서의 force_delete의 사용의 실행과 동일하지만 force_delete()가 가졌던 경쟁을 가지지 않는다.

delete_inode: VFS가 아이노드르를 지우려 할때 때 호출 됨

put_super: VFS가 수퍼블록을 프리(unmount)할 때 호출되며 superblock 락과 함께 호출된다.

write_super: VFS가 superblock을 디스크에 쓰려고 할 때 호출됨, OPTIONAL

sync_fs: VFS가 수퍼블록에 관한 더티 데이터를 쓰려고 할때 호출됨 두 번째 파라미터는 메쏘드가 write out이 끝날 때까지 기다릴지를 가르킨다. OPTIONAL

write_super_lockfs: VFS 가 파일시스템을 락킹할 때 그것을 일관된 상태로 강제할 때 이 함수는 호출된다. 현재 LVM에 의해 사용된다.

unlockfs: 언락킹하고 다시 쓰기 가능한 상태로 만들 때 호출된다.

statfs:VFS가 파일시스템 통계를 요구할 때 호출된다. 커널락이 필요.

remount_fs: 파일시스템 리마운트될 때 이것은 락이 필요.

clear_inode: VFS 가 아이노드를 클리어할 때 호출 된다. OPTIONAL

umount_begin: VFS가 마운트 할 때 호출 됨.

sync_inodes: VFS가 수퍼 블록에 연관된 더티 데이터를 쓰려고 할때 호출됨

show_options: VFS가 마운트 옵션을 보여줄때 호출됨

quota_read: 파일시스템 쿼터 파일을 읽기 위해 호출된다.

quota_write: 파일시스템 쿼터 파일을 쓰려고 할때 호출됨

- Inode Operations (struct inode_operations)

```
struct inode_operations {
int (*create) (struct inode *,struct dentry *,int,
struct   nameidata *);
struct dentry * (*lookup) (struct inode *,struct dentry
*, struct nameidata *);
int (*link) (struct dentry *,struct inode *,struct
dentry *);
int (*unlink) (struct inode *,struct dentry *);
int (*symlink) (struct inode *,struct dentry *,const
char *);
int (*mkdir) (struct inode *,struct dentry *,int);
int (*rmdir) (struct inode *,struct dentry *);
int (*mknod) (struct inode *,struct dentry *,
int,dev_t);
int (*rename) (struct inode *, struct dentry *, struct
inode *, struct dentry *);
int (*readlink) (struct dentry *, char __user *,int);
void * (*follow_link) (struct dentry *, struct
nameidata *);
void (*put_link) (struct dentry *, struct nameidata *,
void *);
void (*truncate) (struct inode *);
int (*permission) (struct inode *, int, struct
nameidata *);
int (*setattr) (struct dentry *, struct iattr *);
int (*getattr) (struct vfsmount *mnt, struct dentry *,
struct kstat *);
int (*setxattr) (struct dentry *, const char *,const
void *,size_t,int);
ssize_t (*getxattr) (struct dentry *, const char *,
void *, size_t);
ssize_t (*listxattr) (struct dentry *, char *, size_t);
};
```

```
struct inode_operations
sfs_dir_inode_operations = {
        .create         = sfs_create,
        .lookup         = sfs_lookup,
        .link           = sfs_link,
        .unlink         = sfs_unlink,
        .mkdir          = sfs_mkdir,
        .rmdir          = sfs_rmdir,
        .mknod          = sfs_mknod,
        .rename         = sfs_rename,
        .getattr        = sfs_getattr,
};

struct inode_operations
sfs_file_inode_operations = {
        .truncate       = sfs_truncate,
        .getattr        = sfs_getattr,
};
```

표 3.8 VFS 아이노드 오퍼레이션과 파일시스템 템플릿 아이노드 오퍼레이션 비교

- 아이노드 오퍼레이션 별 역할

create: open(2) 과 creat(2) 시스템 콜에 의해 호출됨, 레귤러 파일을 지원할 경우에만 필요하다. 아이노드를 가지지 않는 경우의 dentry. negative dentry가 되어야 한다. 이 함수에서 d_instantiate()를 dentry와 새롭게 생성된 아이노드로 호출해야 할 것이다.

lookup: dentry에 있는 이름으로 부모 디렉토리에서 아이노드를 찾는다. 꼭 d_add()함수를 찾아낸 아이노드를 dentry에 삽입하기 위해 호출해야 한다. 아이노드 구조체에 있는 I_count 필드는 증가되어야만 한다 .만일 이름있는 아이노드가 존재 하지 않는 다면 NULL 아이노드는 네거티브 dentry에 삽입되어져야 한다. 실제 에러가 나면 이 루틴에서 에러코드를 리턴해야만 한다 . 그렇지 않은 경우 아이노드를 생성하는 것은 creat(2), mknod(2), mkdir(2) 등등은 실패할 것이다 . dentry 함수를 오버로딩하고 자하면 dentry 의 d_dop 필드를 초기화 하여야 한다. 이것은 디렉토리 아이노드 세마포어와 함께 호출된다.

link: link(2) 시스템콜 에 의해 호출됨. 하드링크를 서포트 하고 자할 때 필요하다. d_instantiate()를 호출해야 할 것다 . create() 처럼

unlink: unlink(2)에 의해 호출된다 아이노드를 지우기를 지원할 경우만 필요하다.

symlink: symlink(2) system call에 의해 호출됨, symlink를 지원하기 원할 때만 필요하다. d_instantiate()를 호출해야 할것이다 create() 처럼

mkdir: mkdir(2) 시스템 콜에 의해 호출됨 서브디렉토리 생성을 지원하려고 할 때만 필요하다. d_instantiate()를 호출해야 할것이다 create() 처럼

rmdir: rmdir(2) 시스템 콜에 의해 호출됨 , 서브디렉토리를 지우고자 할때만 필요함

mknod: mknod(2) 시스템콜에 의해 호출됨 char or block device 아이노드 혹은 FIFO, 소켓을 생성하기 위해, 이런 타입의 아이노드를 생성하려고 할 때만 필요. d_instantiate()를 호출해야 할 것이다 create() 처럼

rename: rename(2) 시스템 콜에 의해 두 번째 아이노드와 dentry에 주어진 부모와 이름을 소유하는 객체의 이름을 재명명하기 위해 호출된다.

readlink: read_link(2)에 의해 호출, 심볼릭 링크를 읽기 위해서만 사용

follow_link: VFS가 심볼릭 링크를 따라가기 위해 호출된다. 심볼릭 링크를 지원할 때만 필요하다. put_link()에게 보내지는 void pointer cooki를 리턴한다.

put_link: follow_link()에 의해 할당된 리소스들을 릴리즈하기 위해 호출된다 follow_link()에 의해 리턴되는 쿠키는 마지막 파라미터로 전달된다. NFS처럼 페이지 캐쉬가 스테이블하지 않은 파일시스템들에 의해 사용된다 .(심볼릭 링크 워크 가 시작되었을 때 설치 되는 페이지는 워크의 마지막 단계에서 페이지 캐쉬에 없을 지도 모른다.)

truncate: 파일의 사이즈을 바꿀때 호출되고 I_sie의 가 이 함수가 호출되기 전 VFS에 의해 세팅된다. truncate(2) 함수와 관계된 기능들에 의해 호출된다.

permission: POSIX-like filesystem. 에 퍼미션을 체크하다.

setattr: 파일의 속성 세팅. chmod(2) 같은 관련된 시스템 콜에 의해 호출된다.

getattr: VFS가 파일속성을 읽기 위해 호출된다 . 이 함수는 stat(2)와 관계된 기능에서 호출된다.

setxattr: setxattr(2) 시스템 콜에 의해 파일을 위해 확장된 속성을 세팅하기 위해 호출됨. 환정된 속성은 아이 노드에 관련된 name:value 쌍이다.

getxattr: getxattr(2) 시스템 콜에 의해 확장된 속성 이름의 값을 검색하기 위해 호출된다.

listxattr: listxattr(2) 시스템 콜에 의해 확장된 속성을 리스팅하기 위해 호출된다

removexattr: removexattr(2) 시스템 콜에 의해 확장된 속성을 파일로부터 지우기 위해 사용된다.

- Address Space Operation (struct address_space_operations)

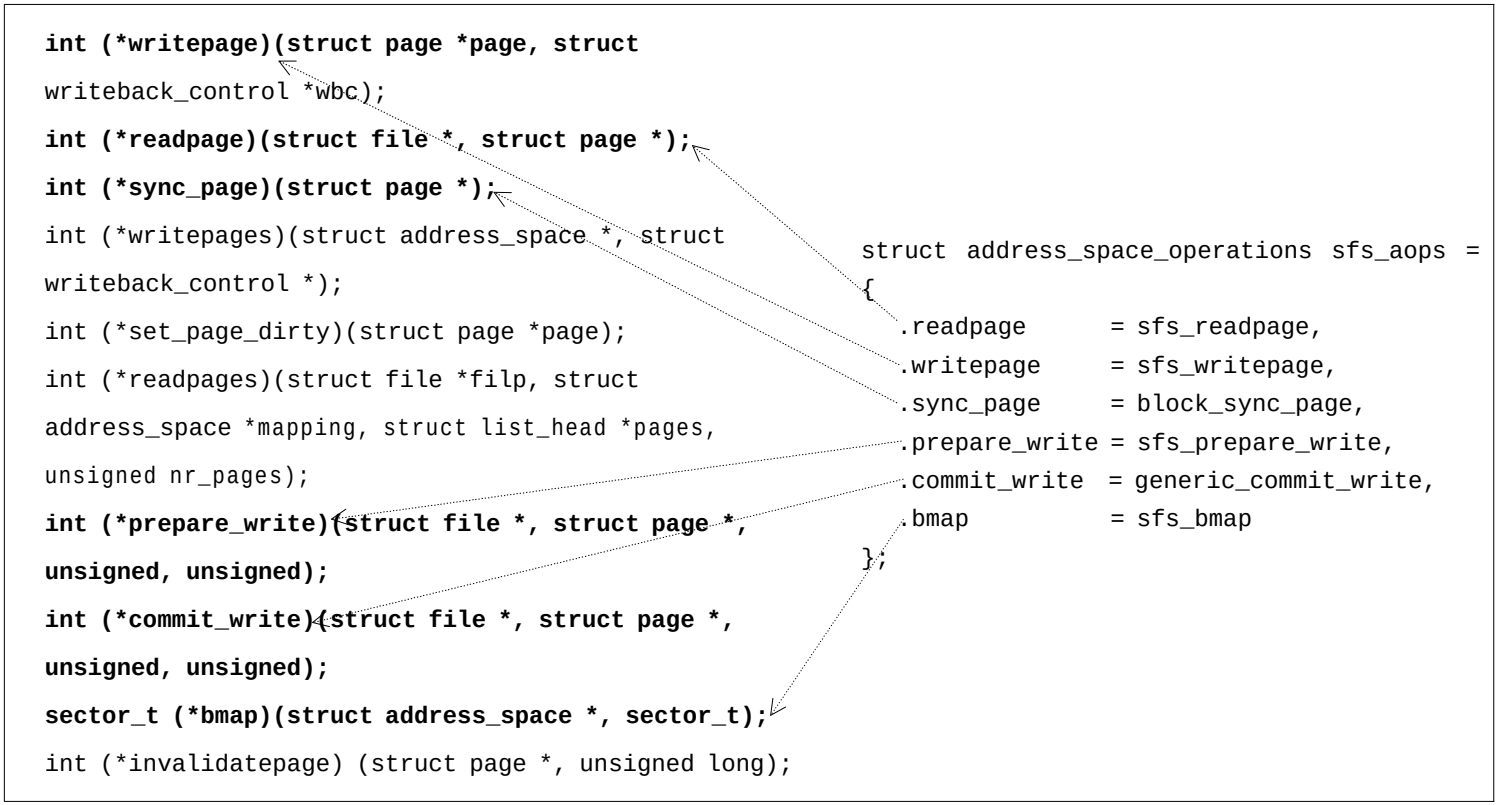

표 3.9 VFS 어드레스 스페이스 오퍼레이션과 파일시스템 템필릿의 어드레스 스페이스 오퍼레이션 비교

- 어드레스 스페이스 오퍼레이션 별 역할

writepage: VM이 더티 페이지를 백킹 스토어로 write 하기 위하여 호출된다.

readpage: VM이 더티 페이지를 백킹 스토어로 read 하기 위하여 호출된다.

sync_page: VM이 더티 페이지를 백킹 스토어에게 대기하고 있는 I/O 함수들을 수행할 것을 알린다.

OPTIONAL

writepages: 어드레스 스페이스 객체에 연관된 page들을 write out 한다.

set_page_dirty: 페이지를 더티 상태로 마크한다.

readpages: readpage와 동일하지만 한 페이지 대신에 여러 페이지를 읽는다. 리드 어헤드(미리읽기)를 위해서 사용된다. 따라서 에러가 나더라도 무시된다.

prepare_write: genric write 시에 호출되고 페이지에 대한 write 요청을 세팅한다. 쓰기 요청을 세팅하면 페이지에 대한 준비가 된다.

commit_write: prepare_write 가 성공하면 새로운 데이터는 페이지로 복사되고 이 함수가 호출된다. 파일의 사이즈를 업데이트하고 아이노드를 더티로 체크한다.

bmap: VFS가 logical block offset을 physical block number 로 매핑하기 위해 호출한다.

releasepage: 페이지를 어드레스 스페이스 객체에서 제거한다.

direct_IO: generic read/write 루틴에서 direct I/O를 수행하기 위해 호출된다. page cache를 건너 뛰고 어드레스 스페이스와 저장장치로 직접 데이터를 전송한다.

- File Object (struct file_operations)

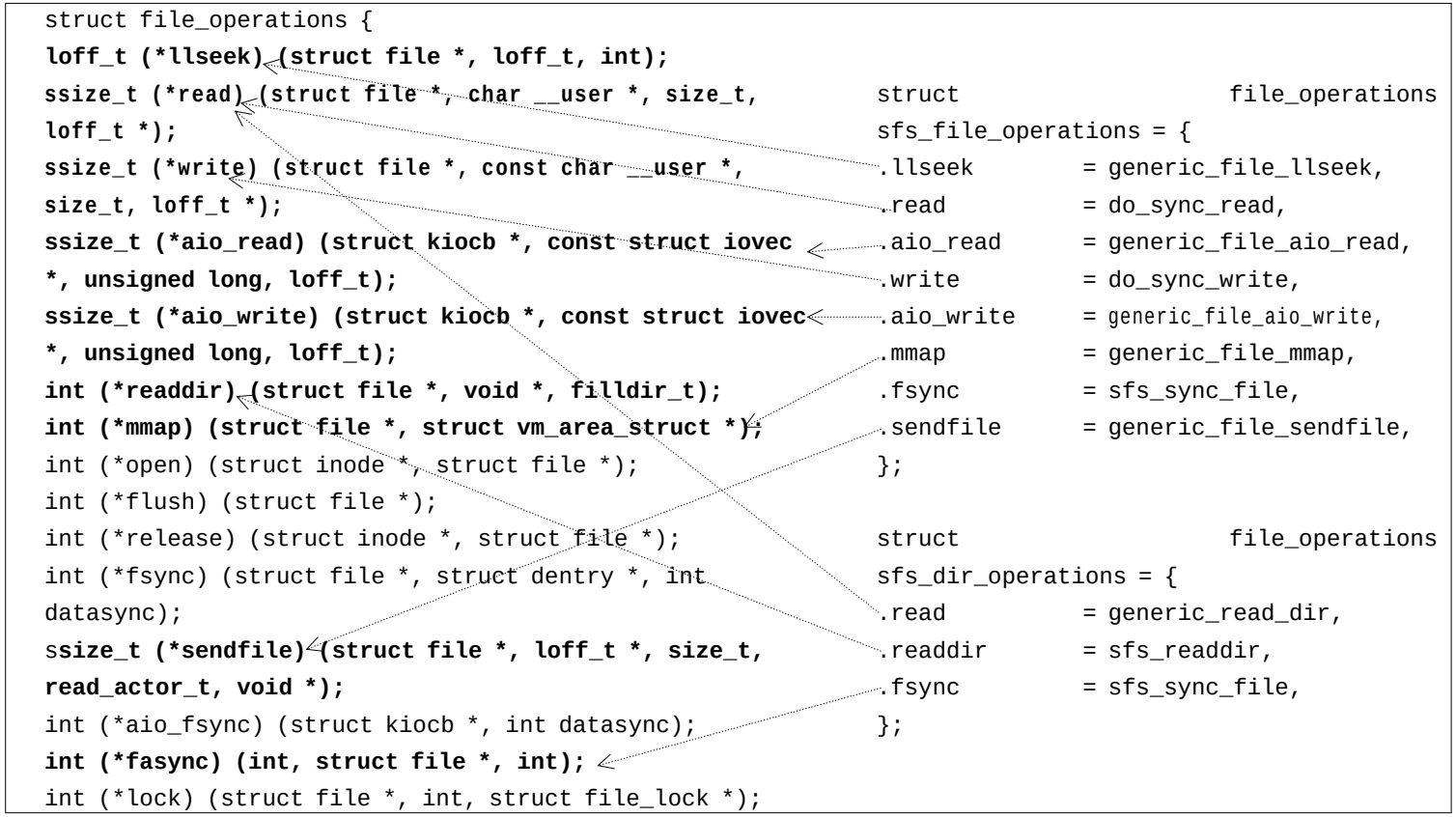

표 3.10 VFS 파일 오퍼레이션과 파일시스템 템플릿 파일 오퍼레이션 비교

- 파일 오퍼레이션 별 역할

read: read(2) 시스템 콜에 의해 호출

aio_read: io_submit(2) 과 비동기 I/O 오퍼레이션에 의해 호출

write: write(2) 시스템 콜에 의해 호출

aio_write: io_submit(2) 에 의해 호출

readdir: VFS가 디렉터리 내용을 읽고자 할 때 호출된다.

poll: 파일에 활동이 있는 지를 VFS가 체크할 때 호출된다.

ioctl: ioctl(2) 시스템 콜에 의해 호출된다.

mmap: mmap(2) 시스템 콜에 의해 호출된다.

open: VFS가 파일을 열 때 호출되고 새로운 “struct file” 객체를 생성한다. 그리고 나서 새롭게 생성된 파일 구조체의 open 함수를 호출한다.

flush: close(2) system call 에 의해 파일을 flush하기 위해 호출된다.

release: 열린 파일에 대한 마지막 레퍼러스가 close 될 때 호출된다.

fsync: fsync(2) system call 에 의해 호출된다.

fasync: 비동기(non-blocking mode) 모드가 활성화 되었을 때 fcntl(2) system call에 의해 호출된다.

lock: fcntl(2)이 F_GETLK, F_SETLK, and F_SETLKW에 대한 명령을 수행할 때 호출된다.

readv: readv(2) system call에 의해 호출된다.

writev: writev(2) system call에 의해 호출된다.

sendfile: sendfile(2) system call에 의해 호출된다.

get_unmapped_area: mmap(2) system call에 의해 호출된다.

check_flags: fcntl(2)이 F_SETFL command를 수행할 때 호출된다.

dir_notify: fcntl(2)이 F_NOTIFY command를 수행할 때 호출된다.

flock: flock(2) system call에 의해 호출된다.

splice_write: called by the VFS to splice data from a pipe to a file. This method is used by the splice(2) system call

splice_read: called by the VFS to splice data from file to a pipe. This method is used by the splice(2) system call

나. Kernel API

VFS의 객체를 다루기 위해서 혹은 커널 프로그래밍을 위하여 Kernel API를 사용해야 한다. Kernel API는 그 문서화가 이루어져 있는 것도 있고 없는 것도 있다. 파일시스템 템플릿 구현 시 사용한 Kernel의 함수들의 목록과 설명을 기술한다. 많은 수의 API들이 존재하지만 VFS에 요구하는 객체와 메쏘드들을 구현하기 위해 필요한 목록은 다음과 같다. 이 목록들은 구현된 파일시스템의 코드 리딩시 참조되어지기 위하여 작성이 되었다. 각 API의 내용은 매뉴얼이 존재하는 경우 그 내용을 번역하였고 없는 경우에는 주석과 함수의 본체를 분석하여 작성하였다. 다음의 내용들은 2.6.20 외의 버전에서는 다를 수 있다. 함수의 기술에서 사용 예로 작성된 코드를 삽입하여 설명하는 이 코드들은 필요한 부분을 강조하기 위해서 축약되어 있다.

Synopsis	void lock_kernel(void);
Arguments	void
return	void

Description	큰 커널 락을 얻기 위하여 호출되고 반드시 unlock 되어야함
	unlock_kernel(void) 함수를 통해서 락을 제거하고 락이 걸리는 영역은 작을 수록 좋다.
	VFS에 관련된 Locking Rule은 커널문서 /Documentation/filesystems/Locking에서 확인할 수 있다. VFS 객체의 모든 함수에 대한 규칙이 명시되어 있다.
	
	사용 예)
	`void minix_free_block(struct inode * inode, int block)`
	`{`
	`  lock_kernel();`
	`  if (!minix_test_and_clear_bit(bit,bh->b_data))`
	`  printk("free_block (%s:%d): bit already cleared\n",`
	`  sb->s_id, block);`
	`  unlock_kernel();`
	`  mark_buffer_dirty(bh);`
	`  return;`
	`}`
	블록을 free시키기 위하여 해당하는 블록의 블록 비트를 clear한다.

Synopsis	void fastcall mark_buffer_dirty (struct buffer_head * bh);
Arguments	bh : 더티 버퍼로 마크할 버퍼헤드 구조체
return	void

Description	버퍼에 더티 비트를 세팅하고 나서 그 버퍼의 페이지를 더티 비트 세팅한다. 그리고 나서 address_space에 더티 페이지로 태깅한다. 대부분의 on-disc 자료구조를 buffer_head로 읽어서 처리하기(bread()함수를 호출하여) 때문에 이 함수를 이용해 커널에게 wrtie-out이 수행되어야 한다고 알린다.
	
	사용 예)
	`static void sfs_put_super(struct super_block *sb)`
	`{`
	`int i;`
	`struct sfs_sb_info *sbi = SFS_SB(sb);`
	
	`if (!(sb->s_flags & MS_RDONLY)) {`
	`sbi->s_fs->s_state = sbi->s_mount_state;`
	`**mark_buffer_dirty(sbi->s_sbh);**`
	`}`
	
	`brelse(sbi->s_bmap);`
	`brelse(sbi->s_imap);`
	`brelse (sbi->s_sbh);`
	`return;`
	
	`}`
	
	디스크의 수퍼블록을 메모리상의 sfs 수퍼블록의 내용으로 갱신하는 코드이다. 직접 write 명령을 내리지 않고 위와 같이 버퍼의 dirty 마크를 하는 것이 전부이다. 커널 쓰레드 pdflush 가 그 역할을 수행한다.

Synopsis	static inline struct buffer_head * sb_bread(struct super_block *sb, sector_t block)
Arguments	sb : 읽을 블록을 포함하고 있는 수퍼블록 block : 읽을 블록 넘버
return	성공 : block에서 지정한 데이터를 포함하는 buffer_head 실패 : NULL (unreadable)

Description	__bread(sb->s_bdev, block, sb->s_blocksize)을 호출하는 랩퍼 함수이다. 지정한 블록 번호로 디바이스에서 읽어오는 함수이다. 파일시스템에서 메타데이터(수퍼블록)을 읽을 때 사용한다. block 단위(파일시스템에서 지정한)로 읽기 때문에 페이지를 사용하지 않는다. 사용이 끝난 후에는 반드디 brelse()함수를 호출하여 해제하여야 한다.
	
	사용 예)
	`static int sfs_fill_super(struct super_block *s, void *data, int silent)`
	`{`
	`struct buffer_head *bh;`
	`struct sfs_super_block *ss;`
	`long i, block;`
	
	`if (!sb_set_blocksize(s, SFS_BLOCK_SIZE))`
	`goto out_bad_hblock;`
	
	`if (!(bh = sb_bread(s, 1)))`
	`goto out_bad_sb;`
	
	sfs_fill_super()함수는 파일시스템 메터데이터를 읽기 위하여 블록단위 READ한다. 위의 코드는 1번 블록(1KB단위)를 디스크로부터 읽어오는 코드이다.

Synopsis	void clear_inode (struct inode * inode);
Arguments	inode : 클리어 할 아이노드
return	void

Description	더 이상 이 아이노드가 필요없음을 나타낸다.
	(clear의 의미는 그 아이노드를 더 이상 참조하지 않기 때문에 free 될수 있다는 것을 의미한다.)
	
	사용 예)
	`void sfs_free_inode(struct inode * inode)`
	`{`
	`sfs_clear_inode(inode);`
	`lock_kernel();`
	`if(!test_and_clear_bit(ino-1, (unsigned long *)bh->b_data)) {`
	`sfs_debug("inode %d bit already cleared\n", ino);`
	`}`
	`unlock_kernel();`
	`mark_buffer_dirty(bh);`
	
	`sbi->s_fs->s_free_inodes_count++;`
	`mark_buffer_dirty(sbi->s_sbh);`
	
	`clear_inode(inode);` `// fs/inode.c`
	`}`
	
	아이노드를 프리시키는 함수이고 여기서 파일시스템 specific 한 작업을 하고 VFS에 아이노드를 clear한다. 이 함수에는 아이노드의 i_state를 I_CLEAR 상태로 만든다.

Synopsis	static inline void insert_inode_hash(struct inode *inode)
Arguments	inode: 아직 해시테이블에 삽입되지 않은 상태의 아이노드
return	void

Description	inode를 해시 한다. 수퍼블록에 대한 아이노드 해시 테이블에 아이노드를 삽입한다.
	
	사용 예)
	`struct inode * sfs_new_inode(const struct inode * dir, int * error)`
	`{`
	
	`  inode->i_uid = current->fsuid;`
	`  inode->i_gid = (dir->i_mode & S_ISGID) ? dir->i_gid : current->fsgid;`
	`  inode->i_ino = i+1;`
	`  inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME_SEC;`
	`  inode->i_blocks = 0;`
	`  memset(&SFS_I(inode)->i_data, 0,`
	`  sizeof(SFS_I(inode)->i_data));`
	`  **insert_inode_hash(inode);** // fs.h fs/inode.c`
	`  mark_inode_dirty(inode);// fs.h fs/fs-writeback.c`
	
	아이노드를 새로 생성할 때 이 함수는 반드시 실행되어야 하고 이 함수를 통해 VFS가 유지하는 해시 테이블에 새로 생성한 아이노드가 유지된다.

Synopsis	static inline void mark_inode_dirty(struct inode *inode)
Arguments	inode : 더티로 세팅할 inode
return	void

Description	아이노드를 더티로 세팅한다. 해시가 되기 전에는 절대로 dirty list에 옮겨지지 않으므로 이 함수를 호출하기 전에 해시를 하여야 한다. ATOMIC 함수이므로 spinlock이 필요하다.
	
	사용 예)
	`struct inode * sfs_new_inode(const struct inode * dir, int * error)`
	`{`
	
	`  inode->i_uid = current->fsuid;`
	`  inode->i_gid = (dir->i_mode & S_ISGID) ? dir->i_gid : current->fsgid;`
	`  inode->i_ino = i+1;`
	`  inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME_SEC;`
	`  inode->i_blocks = 0;`
	`  memset(&SFS_I(inode)->i_data, 0,`
	`  sizeof(SFS_I(inode)->i_data));`
	`  insert_inode_hash(inode); // fs.h fs/inode.c`
	`  **mark_inode_dirty(inode);**// fs.h fs/fs-writeback.c`
	
	아이노드를 새로 생성한 바로 그 상태에서는 아이노드의 내용이 디스크에 반영되지 않은 상태이다. 그래서 디스크에 반영될 것을 요청하기 위하여 이 함수를 호출한다.

Synopsis	void iput (struct inode * inode);
Arguments	inode : 내보낼 inode
return	void

Description	사용 count를 감소시킨다. 사용 count가 0이라면 이 아이노드는 후에 free 되고 파괴될 수 있다.
	
	사용 예)
	`struct inode * sfs_new_inode(const struct inode * dir, int * error)`
	`{`
	`struct super_block *sb = dir->i_sb;`
	`struct sfs_sb_info *sbi = SFS_SB(sb);`
	`struct inode *inode = new_inode(sb); // fs/inode.c`
	`struct buffer_head *bh = NULL;`
	`int i = 0;`
	`if (!i || i > SFS_BLOCK_SIZE/sizeof(struct sfs_inode)-1) {`
	`goto out;`
	`}`
	`out:`
	`unlock_kernel();`
	`iput(inode);`
	`return NULL;`
	`}`
	파일시스템 디스크 아이노드를 어떤 이유로 할당이 불가능할 때 VFS 아이노드는 이미 생성한 상태이므로 해제하여만 한다. 그 경우 iput() 을 사용한다.

Synopsis	static int filldir(void * __buf, const char * name, int namlen, loff_t offset, u64 ino, unsigned int d_type)
Arguments	name : 엔트리 이름 namelen : 엔트리 이름의 길이 offset : 엔트리 수 만큼 증가 ino : 엔트리가 가르키는 아이노드 번호 d_type : 파일타입
return	성공 : 0

Description	readdir()함수에 의해 사용되고 커널이 요구하는 dirent의 명시한다. 커널이 디렉터리를 커널의 스페이스로 읽어올 수 이겠다한다. 각 파일시스템의 다른 형테의 dirent구조를 가지고 있으므로 directory를 읽은 후에 이 함수를 통해 커널에 그 엔트리를 전달하면 커널에서는 readdir()함수에 대한 요청이 올 때 그 엔트리를 유저 영역에 전달할 수 있게 된다.
	
	사용 예)
	`static int sfs_readdir(struct file * filp, void * dirent, filldir_t filldir)`
	`{`
	
	`  if (de->d_ino) {`
	`  over = filldir(dirent, de->d_name, l,`
	`  filp->f_pos, //(0<<PAGE_CACHE_SHIFT)`
	`  | offset,de->d_ino, DT_UNKNOWN);`
	`  if (over) {`
	`  dir_put_page(page);`
	`  goto done;`
	`  }`
	`  }`
	`  }`
	
	`  return 0;`
	`}`
	dir에 대한 read 요청시 여러가지 작업후에 VFS에 dentry를 추가하는 작업이 필요하다. 그 때 filldir()함수에 위와 같이 dentry에 적합한 정보를 인자로 주어 호출한다.

Synopsis	int sync_mapping_buffers (struct address_space * mapping);
Arguments	mapping : 버퍼들이 쓰여지길 원하는 address_space
return	

Description	sync_mapping_buffers - write out and wait upon a mapping한 매핑에 쓰기(write out)를 하고 기다린다.
	기본적으로 fsync 함수에 대한 편의함수이고 매핑은 성공적으로 fsync 되기 위하여 필요한 버퍼들에 대한 파일 혹은 디렉터리이다.
	
	사용 예)
	`int sfs_sync_file(struct file * file, struct dentry *dentry, int datasync)`
	`{`
	`  struct inode *inode = dentry->d_inode;`
	`  int err;`
	
	`  err = sync_mapping_buffers(inode->i_mapping);`
	`  // fs/buffer.c`
	`  if (!(inode->i_state & I_DIRTY))`
	`  return err;`
	
	`  err |= sfs_sync_inode(inode); // inode.c`
	`  return err ? -EIO : 0;`
	`}`
	
	파일에 대한 쓰기 정보는 mapping(address_space 객체) 가 소유하고 있으므로 이 매핑을 싱크함으로써 파일 싱크 작업이 이루어진다.

Synopsis	struct kmem_cache *kmem_cache_create (const char *name, size_t size, size_t align, unsigned long flags, void (*ctor), void (*dtor))
Arguments	name : /proc/slabinfo 에서 식별하기 위해 사용되는 이름 size : 캐시에서 생성될 객체의 크기 align : 객체에 대해 요구되는 메모리 정렬 flags : SLAB flags ctor : 객체들에 대한 생성자 dtor : 객체들에 대한 소멸자
return	성공 : kmem_cache 에 대한 포인터 실패 : NULL

Description	캐시를 생성한다. 성공시에는 캐시에 대한 ptr 을 리턴하고 실패시에는 NULL 리턴한다. 생성자는 캐시에 의해 페이지가 할당되어질 때 실행되고 소멸자는 페이지를 돌려지기 전에 실행된다.
	name : 캐시가 파괴되기 전까지 유효해야 한다.
	이 함수를 호출하는 모듈이 언로드 되기 전에 반드시 파괴되어야 한다.
	Slab Flags
	SLAB_POISON
	SLAB_RED_ZONE
	
	사용 예)
	`static int init_inodecache(void)`
	`{`
	`sfs_inode_cachep =`
	`kmem_cache_create("sfs_inode_cache",`
	`sizeof(struct sfs_inode_info), 0,`
	`(SLAB_RECLAIM_ACCOUNT|SLAB_MEM_SPREAD),`
	`init_once, NULL);`
	
	파일시스템 초기화시 그 파일시스템을 위한 아이노드 캐시를 생성하는 데 이 함수가 그 역할을 한다.

Synopsis	void kmem_cache_destroy (struct kmem_cache * cachep);
Arguments	cachep : 파괴할 cache
return	void

Description	slab cache로 부터 struct kmem_cache를 제거한다. 모듈이 언로드 될 때 호출되어져야 한다. 캐쉬를 지우고 모듈이 로드되고 언로드 되는 사이에 중복된 캐시를 피한다.
	
	사용 예)
	`static void destroy_inodecache(void)`
	`{`
	`  kmem_cache_destroy(sfs_inode_cachep);`
	`}`
	
	kmem_cache_creat의 counterpart로서 그 반대의 작업을 위해 파일시스템이 등록 해제 되기 전에 아이노드 캐시를 파괴한다.

Synopsis	struct dentry * d_alloc_root (struct inode * root_inode);
Arguments	root_inode : 루트 아이노드로 할당할 아이노드
return	성공 : 초기화된 아이노드에 대한 엔트리 실패 : NULL

Description	root("/") dentry를 주어진 아이노드에 대해 할당한다. 아이노드는 초기화된다. 메모리가 충분치 않거나 inode가 NULL로 패스 되었을 때 NULL이 리턴된다.
	
	사용 예)
	`static int sfs_fill_super(struct super_block *s, void *data, int silent)`
	`{`
	`  s->s_op = &sfs_sops;`
	`  root_inode = iget(s, 1);`
	`  if (!root_inode)`
	`  goto out_no_root;`
	`  if (!root_inode)`
	`  goto out_no_root;`
	
	`  s->s_root = d_alloc_root(root_inode);`
	`  if (!s->s_root)`
	`  goto out_iput;`
	
	최초의 파일시스템에 등록되고 마운트 될 때 루트 디렉터리 / 를 dentry 로서 VFS에 추가하여야 한다. 이 함수는 루트 아이노드를 dentry를 / 의 이름으로 초기화 하여 생성하고 연결한다. 초기화 루틴이므로 한 번만 호출한다.

Synopsis	static inline void map_bh(struct buffer_head *bh, struct super_block *sb, sector_t block)
Arguments	bh : 블록번호와 사이즈를 buffer_head sb : 장치에 대한 수퍼블록 block : 설정할 블록 넘버
return	void

Description	I/O 에 대상이 되는 블록을 찾은 후 그에 대한 설정을 bh에 대해 하기 위한 함수. 2.4에서는 함수화 되지 않았으나 2.6에서는 함수화 하였고 get_block함수에서 호출하여야 한다.
	
	사용 예)
	`inline int get_block(struct inode * inode, sector_t block, struct buffer_head *bh, int create)`
	`{`
	`directblock_avail:`
	`blk = sfs_new_block(inode);`
	
	`blk = sfs_find_blk(inode, block);`
	
	`if(blk < 0) {`
	`err = blk;`
	`goto out_no_block;`
	`}`
	
	`map_bh(bh, sb, blk);`
	`unlock_kernel();`
	`return 0;`
	`}`
	
	적합한 블록 번호를 찾고 나서 VFS에 buffer_head 구조체를 갱신해서 알려주어야 한다. 이 함수는 bh->b_blocknr을 갱신한다.

Synopsis	void d_instantiate (struct dentry * entry, struct inode * inode);
Arguments	entry : 필드를 채울 dentry inode : 엔트리에 연결할 inode
return	void

Description	엔트리에 아이노드 정보를 채운다.
	
	사용 예)
	`static int add_nondir(struct dentry *dentry, struct inode *inode)`
	`{`
	`int err = sfs_add_link(dentry, inode);`
	
	`if (!err) {`
	`d_instantiate(dentry, inode);`
	`// fs/dcache.c fill in inode info for a dentry`
	`return 0;`
	`}`
	
	`inode_dec_link_count(inode);`
	`iput(inode);`
	`return err;`
	`}`
	
	디렉토리 항목이 새로 생성될 때 dentry 생성시에 필요한 함수로 파일시스템 관점에서는 디렉토리의 엔트리가 증가할 때마다 호출하여야 한다.

Synopsis	static inline struct inode *iget(struct super_block *sb, unsigned long ino)
Arguments	sb : 파일시스템의 수퍼블록 ino : 디스크로부터 가져올 아이노드 번호
return	성공 : 존재하는 아이노드 혹은 새로운 아이노드 실패 : NULL (NO MORE INODE)

Description	아이노드 넘버로 아이노드 캐시에서 아이노드를 찾고 아이노드가 존재한다면 참조 카운트를 증가시키고 그 아이노드를 리턴한다. 캐시에 아이노드가 존재하지 않으면 새 아이노드를 할당하고 I_NEW flag로 세팅하고 리턴한다. 그리고 나서 파일시스템의 read_inode()를 호출하여 아이노드의 정보를 채운다.
	
	사용 예)
	`static int sfs_fill_super(struct super_block *s, void *data, int silent)`
	`{`
	`  s->s_op = &sfs_sops;`
	`  root_inode = iget(s, 1);`
	`  if (!root_inode)`
	`  goto out_no_root;`
	`  if (!root_inode)`
	`  goto out_no_root;`
	
	`  s->s_root = d_alloc_root(root_inode);`
	`  if (!s->s_root)`
	`  goto out_iput;`
	
	아이노들을 생성하기 위해서 가용한 아이노드를 찾아 할당하기 위해 호출하는 데 이 함수 안에서 위에서 설명한 것과 같이 sfs_read_inode를 호출하여 각종 오퍼레이션을 연결한다.

Synopsis	void d_add (struct dentry * entry, struct inode * inode);
Arguments	entry : 추가할 디렉터리 엔트리 inode : 이 엔트리에 연결할 아이노드
return	void

Description	해시큐에 엔트리를 연결하고 아이노드를 초기화한다. 엔트리는 실제로 d_alloc() 에서 초기화 되어있는 상태이다..
	
	사용 예)
	`static struct dentry *sfs_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)`
	`{`
	
	`if(ino) {`
	`inode = iget(dir->i_sb, ino);`
	`if(!inode)`
	`return ERR_PTR(-EACCES);`
	`}`
	
	`d_add(dentry, inode);`
	`// fs/dcache.c add dentry to hash Q`
	`return NULL;`
	`}`
	
	디렉토리 항목을 찾는 데 성공하였을 때 dentry를 생성하고 hash 테이블에 삽입한다. 이 함수는 d_instantiate()를 호출한다.

3.3 파일시스템 템플릿 구현

3.3.1 개발환경

커널개발은 일반적으로 LKM(Loadable Kernel Module)로 개발하여 커널 전체를 컴파일을 피한다. 파일시스템 역시 모듈로 개발이 가능하며 본 연구에서도 같은 방식으로 개발한다. 커널은 C와 어셈블리로 이루어져 있으며 본 템플릿은 C로 구현하였다. 테스트 및 디버깅을 위해서 개발 머신과 타겟 머신이 필요하고 본 연구에서는 타겟 머신을 가상화하여 실험환경을 구축하였다. 파일시스템이 사용될 디스크 디바이스 또한 호스트의 파일로서 가상화한다.

- 개발 머신

CPU	Intel Pentium4 1.7
OS	Fedora Core 4 2.6.11
Compiler	gcc 4.0
Kernel Ver(개발용)	2.6.20

표 3.11 개발 환경

- 타겟 머신(User-Mode-Linux)

CPU	Virtual
OS	Virtual
Compiler	N/A
Kernel Ver(개발용)	2.6.20

표 3.12 타겟 환경

위의 환경에서 개발 및 테스트가 이루어지고 전체 개발 환경은 이것 외에도 소스 버전 컨트롤, 버그 리포팅, 개발툴들로 구성되나 버그 리포팅은 1인 개발에서는 실용적이지 않아 제외하였다.

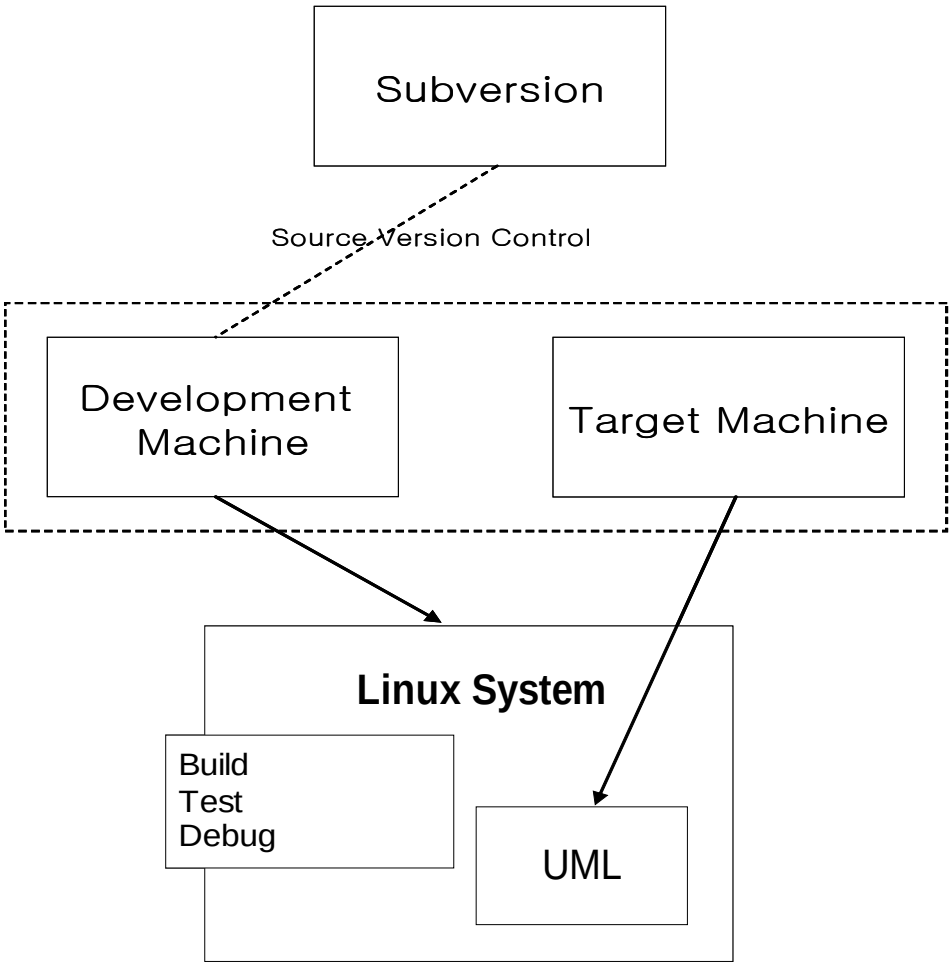

그림 3.8 개발 프로세스

개발 머신에서 코드 작성, 빌드를 하고 테스트 및 디버깅은 타겟 머신에서 이루어진다. 일반적인 커널개발에서는 2개의 머신을 널 모뎀으로 연결하여 하는 작업을 User-Mode-Linux를 사용하면 타겟머신을 가상화하여 한 시스템에서 개발환경 구축이 가능하고 이것은 실험 시에 발생할 수 있는 재부팅 상황(시스템 크래시)을 아주 빠르게 복구할 수 있고, 호스트의 파일을 루트 파일시스템으로 사용하기 때문에 시스템이 완전히 망가질 이유가 전혀 없다. 이로 인해 생산 속도의 향상을 가져온다.

가. UML로 파일시스템 개발환경 만들기

http://user-mode-linux.sourceforge.net

user-mode-linux가 유지 보수 되고 있는 사이트

1. Build UML

Build from source

Kernel hacking with gdb

빌드

```
----------------------------------------------------------
1041  cd linux-2.6.20.3/
1043  make defconfig ARCH=um     // 호스트와 분리되게 컴파일하게 해줌
1045  make menuconfig ARCH=um   // 디버깅 옵션 설정

1048  vi .config                        // 설정 확인
1049  make ARCH=um                    // compile

1050  make modules ARCH=um         // module compile
1051  mount uml/FedoraCore5-x86-root_fs mnt -o loop
1055  make modules_install INSTALL_MOD_PATH=/mnt ARCH=um

1057  cp linux ./uml/
1058  umount /mnt
1059  cd uml/
----------------------------------------------------------
```

UML 부팅

1061 ./linux ubda=FedoraCore5-x86-root_fs mem=256m

udba : 루트 파일시스템

mem : 사용할 메모리(물리적인 메모리를 의미하지 않는다)

2. Filesystem mount / umount (as a host file)

아래는 ext2 파일시스템으로 실험

UML에서 마운트가 문제없이 되며 파일 생성, 저장도 된다.

mkfs util을 통해 disk layout을 쓰는 작업이 선행적으로 요구됨

호스트에서 파일생성, UML 블록 디바이스 추가 옵션

1064 dd if=/dev/zero bs=1M count=100 > ext2_fs

1065 ls

1068 mke2fs ext2_fs

1069 ./linux ubda=FedoraCore5-x86-root_fs ubdb=ext2_fs mem=256m

3. load / unload module

host에서 개발이 이루어지므로 모듈 빌드 후 host filesystem을 마운트해서 host 에

생성된 모듈을 적재한다.

Host file access

101 cat /proc/filesystems // hostfs 지원 여부 확인

103 mkdir hostfs

106 mount none hostfs -t hostfs (-o 특정 디렉터리 or / by default)

107 cd hostfs/

110 cd root/linux-2.6.20.3/uml/

112 cd module_test/

114 insmod hello.ko

115 lsmod

116 rmmod hello.ko

121 cat /proc/kmsg &

122 insmod hello.ko

123 lsmod

124 rmmod hello

4. debugging

커널의 디버깅 : 응용프로그램 디버깅과 완전히 동일하다.

module debugging : 커널디버깅과 유사하나 심벌정보를 추가하여야 한다.

섹션 정보 알아내기

cat /proc/modules | grep sfs

or

cat /sys/moudles/sfs/sections/.text

or

gdb ./linux `cat ~/.uml/UMID/pid`

(gdb) p modules

$4 = {next = 0x1207a044, prev = 0x1207a044}

(gdb) p *((struct module *)0x1207a040)

...

init = 0x1207c000, module_init = 0x0, **module_core = 0x12074000**, init_size =

0, core_size = 28576, init_text_size = 0, core_text_size = 13685, unwind_info = 0x0, arch ...

(gdb) add-symbol-file ../sfs_module/sfs.ko 0x12074000

add symbol table from file "../sfs_module/sfs.ko" at

.text_addr = 0x12074000

(gdb) list super.c:1

Line 1 of "fs/ext2/super.c" is at address 0x81c2638 <ext2_error> but contains no code.

(gdb) b generic_shutdown_super

Breakpoint 2 at 0x80fcef9: file fs/super.c, line 287.

(gdb) c

HOST 시스템에서 프로세스 정보를 보았을 때 UML은 위의 그림처럼 프로세스로서 동작하고 있다. 따라서 프로세스를 디버깅하는 것과 동일하게 리눅스 커널을 디버깅할 수 있다.

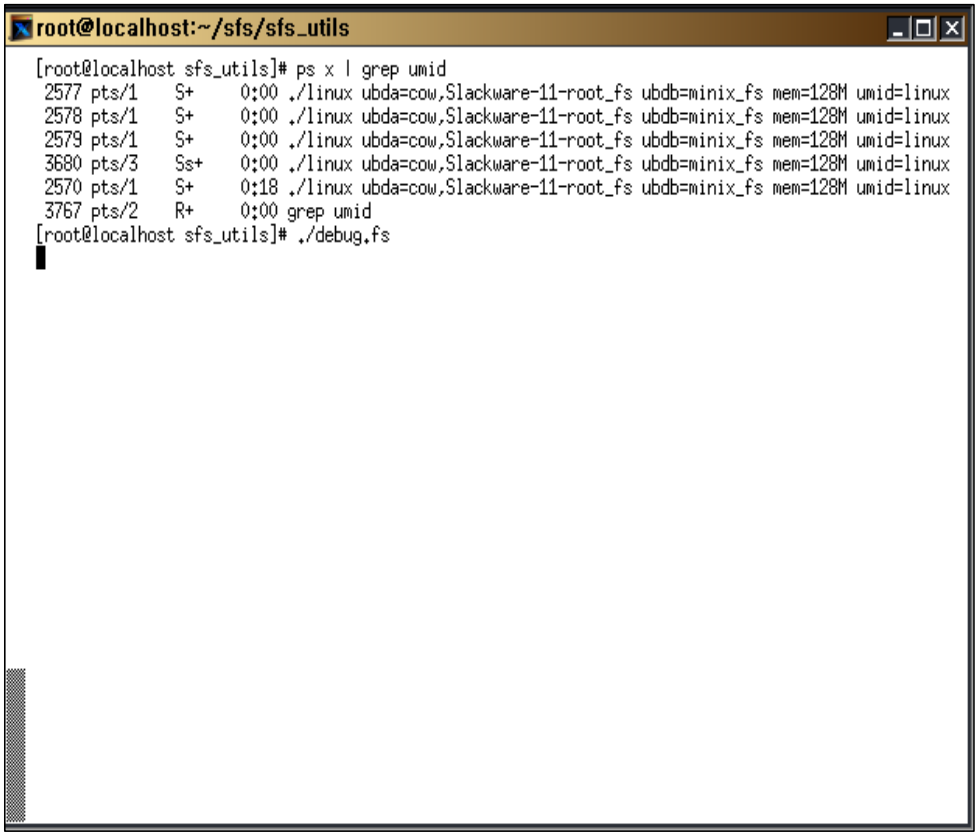

그림 3.11 HOST 시스템에서의 UML

파일시스템 템플릿 코드 중 sfs_fill_super() 에 브레이크을 걸고 파일시스템 마운트 명령을 주었을 sfs_fill_super() 함수에서 대기하고 있는 화면이다.

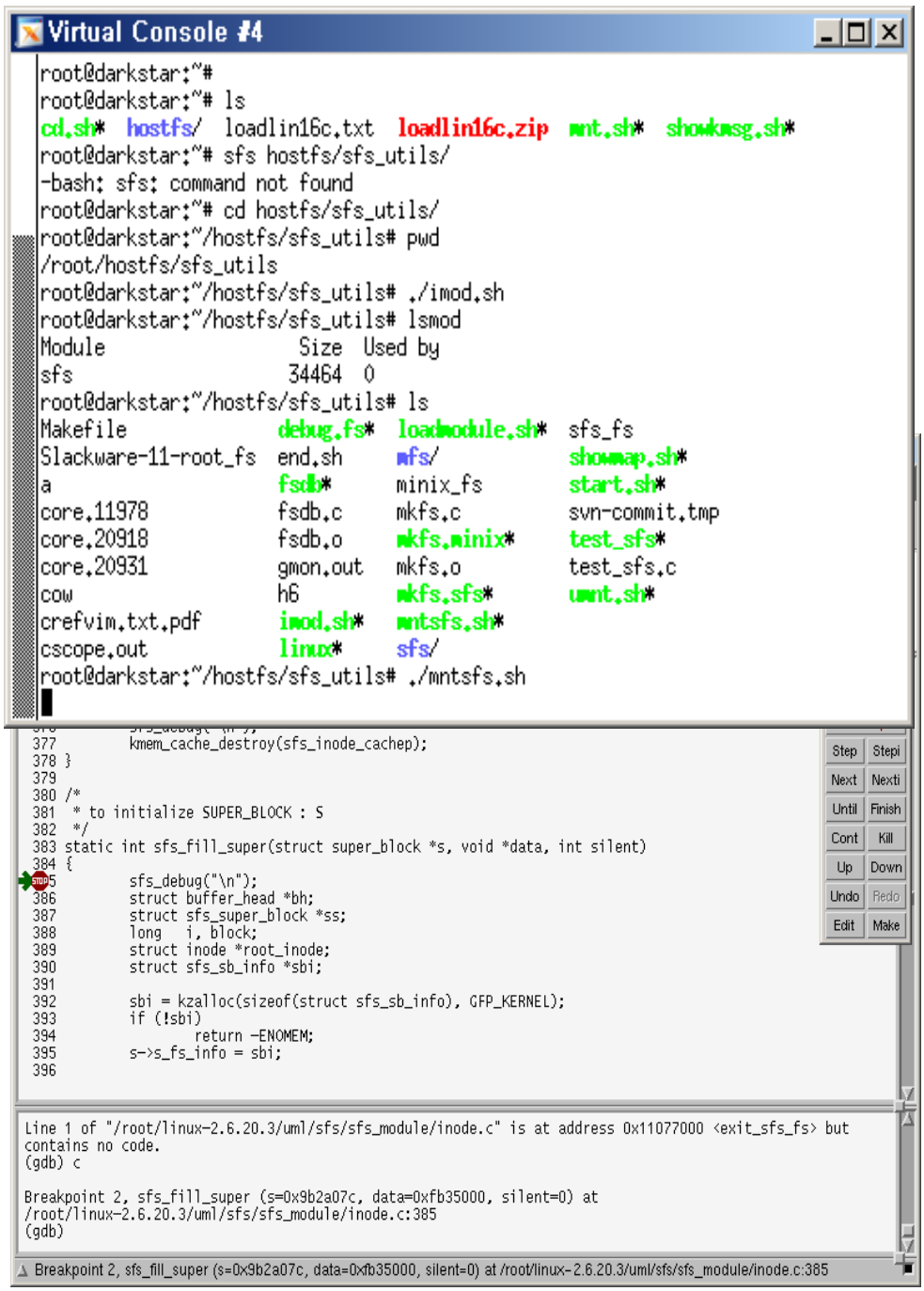

그림 3.12 디버깅 화면

3.3.2 파일시스템 유틸리티의 구현

파일시스템 유틸리티는 3가지로 파일시스템 초기화, 파일시스템 디버깅, 파일시스템 테스트의 역할들로 구성된다.

가. 파일시스템 초기화

파일시스템 초기화란 파일시스템의 레이아웃을 디스크에 쓰는 작업이고 MS-DOS의 format 과 같은 개념이다. 이것을 함으로써 디스크는 파일시스템으로서 사용될 준비가 되는 것이다. 초기화를 함으로써 기존의 정보는 더 이상 유효하지 않게 된다. 본 파일시스템에서는 mkfs.sfs 라는 이름의 초기화 프로그램을 구현했다. (mkfs.sfs) 파일시스템 레이아웃에 따라 다르게 작성하여야 하나 이 프로그램은 기존의 다른 파일시스템들의 초기화 프로그램을 축소하여 작성되었고 템플릿으로 사용 가능하다.

나. 파일시스템 디버거

파일시스템의 초기화가 성공적으로 이루어졌나를 검증하기 위한 프로그램(fsdb.sfs)으로 superblock, inode 등 메타 데이터의 정보를 유저레벨에서 확인할 수 있다.

다. 파일시스템 테스트

파일시스템의 동작을 응용프로그램 레벨에서 테스트 하는 프로그램(test.sfs)를 이용한다.

3.3.3 파일시스템 모듈의 구현

커널 모듈로 파일시스템 템플릿 자체이다. 객체와 그 메쏘드로 구성되어

있고 파일시스템이 초기화 된 이후에 사용이 가능하다. 모듈이 본 연구의 핵심 코드로서 상세한 설명을 위해 각 소스파일 별로 나누어 그 역할에 대해 설명한다.

소스트리 구조는 다음과 같다.

sfs /include/s_fs.h

/sfs_module/bitmap.c

dir.c

file.c

inode.c

itree.c

- s_fs.h : 모듈과 유틸리티의 헤더 파일

파일시스템에서 사용할 매크로들과 데이터 스트럭쳐, 함수의 정의가 작성되어있다. 여기서 파일시스템의 제한 사항을 기술하고 객체를 어떻게 구성할 것인지를 정의한다.

- inode.c

모듈의 등록이 이루어지는 코드가 작성되어 있다. 커널 모듈이 그 자체가 프로그램의 전체기능을 수행하는 것이 아니고 일부로 동작하기 때문에 커널에 등록하는 과정이 필요하다. 그 부분을 이 파일에서 처리한다. 모듈의 등록과 해제을 하고, 수퍼블록 오퍼레이션과 어드레스 스페이스 오퍼레이션이 구현되어 있다.

- namei.c

디렉터리 아이노드에 관한 아이노드 오퍼레이션이 구현 되어 있다.

- bitmap.c

아이노드와 블록에 대한 비트맵처리가 구현 되어 있다. 사용유무를 확인하고 사용할 bit에 대해 mark를 하거나 해제될 bit에 대해 unmark한다.

- dir.c

dir에 대한 파일 오퍼레이션이 구현되어 있다. readdir() 함수에 대한 구현이 되어 있다.

- file.c

file에 관한 파일 오퍼레이션과 아이노드 오퍼레이션이 구현 되어 있다. 파일오퍼레이션은 generic 펑션 즉 vfs에서 제공하는 함수들을 사용하기에 그 구현이 간단하다. 파일에 관한 I/O 역시 page단위로 이루어지기 때문에 address space 객체(inode.c)에서 처리한다.

- itree.c

inode의 블록 포인터에 대한 구현이 이루어진다. 본 템플릿에서는 다이렉트 블록만을 사용하기 때문에 간단하지만 간접 블록을 사용하고자 한다면 이 파일에 그 구현을 추가하여야 한다.

3.3.3 파일시스템 모듈에서 작성한 함수

- inode.c

line 035 : sfs_get_block()

Disc I/O를 할 때 적합한 Logical Block Address 를 찾는다.

line 041 : sfs_writepage() : address_space operation

page에 데이터를 write 한다.

line 046 : sfs_readpage() : address_space operation

page에 있는 데이터를 read한다.

line 051 : sfs_preapare_write() : address_space operation

데이터를 디스크로 쓰기 위해 준비한다. (ll_rw_block)

line 057 : sfs_bmap() : address_space operation

블록 번호를 매핑한다.

line 079 : sfs_getattr()

파일의 속성을 읽는다.

line 086 : sfs_alloc_inode() : super block operation

아이노드를 할당한다.

line 097 : sfs_destroy_inode() : super block operation

inode cache에서 아이노드를 free한다.

line 103 : sfs_read_inode() : super block operation

디스크 아이노드를 메모리로 읽어 온다.

line 139 : sfs_update_inode() : part of super block operation

메모리상의 아이노드의 값을 디스크에 반영한다.

line 169 : sfs_write_inode() : super block operation

아이노드를 write한다.(sfs_update_inode 호출)

line 177 : sfs_sync_inode()

파일오퍼레이션의 일부로 파일 싱크시 호출된다.

line 195 : sfs_delete_inode() : super block operation

아이노드를 제거한다.

line 212 : sfs_put_super() : super block operation

super block을 release 한다.

line 237 : sfs_statfs_inode() : super block operation

파일시스템의 통계 정보를 읽는다.

line 254 : sfs_remount_inode() : super block operation

이미 마운트 되어 있는 파일시스템을 옵션을 바꾸어 마운트한다.

line 298 : init_once() : 파일시스템 초기화 루틴의 일부

아이노드 캐시를 생성한다.

line 320 : destroy_inodcache() : 파일시스템 해제 루틴의 일부

아이노드 캐시를 SLAB 할당자로 부터 제거한다.

line 328 : sfs_fill_super() : super block operation

디스크 수퍼블록을 읽는다.

line 447 : sfs_get_sb : 파일시스템 초기화 루틴의 일부

파일시스템이 마운트 될 때 수퍼블록을 읽는다.

line 599 : init_sfs_fs() : 파일시스템 초기화 루틴

파일시스템을 등록한다.

line 620 : exit_sfs_fs() : 파일시스템 해제 루틴

파일시스템 등록을 해제한다.

- namei.c

line 024 : add_nondir()

디렉토리 데이터 블록의 내용에 디렉토리 엔트리를 추가한다.

line 039 : sfs_create() : dir inode operation

디스크의 아이노드를 생성한다.

line 046 : sfs_lookup() : dir inode operation

dir에서 주어진 이름의 항목을 찾는다.

line 075 : sfs_unlink() : dir inode operation

link count를 감소시키고 디렉토리에서 엔트리를 제거한다.

line 103 : sfs_mkdir() : dir inode operation

디렉토리를 생성한다.

line 151 : sfs_rmdir() : dir inode operation

디렉토리를 제거한다.

line 167 : sfs_mknod() : dir inode operation

노드(파일,디렉토리,장치)를 생성한다.

- bitmap.c

line 24 : count_free()

bitmap에서 가용(free)한 bit를 센다.

line 42 : sfs_free_block()

블록을 해제한다.

line 62 : sfs_new_block()

블록을 할당한다.

line 92 : count_free_blocks()

가용한 블록을 여러 개의 block bitmap에서 합산한다.

line 98 : sfs_raw_inode()

디스크 아이노드를 메모리로 읽어 온다.

line 24 : sfs_clear_inode()

디스크 아이노드를 제거한다.

line 132 : sfs_free_inode()

아이노드를 해제한다.

line 156 : sfs_new_inode()

아이노드를 할당한다.

line 206 : sfs_count_free_inodes

가용한 아이노들을 여러 개의 inode bitmap에서 합산한다.

- dir.c

line 029 : dir_put_page()

디렉터리 엔트리 page를 반환한다.

line 035 : sfs_next_entry()

다음 디렉터리 엔트리를 찾는다.

line 042 : sfs_readdir : dir file operation

디엔트리 객체에 존재하는 디렉터리 엔트리를 전부 읽는다.

line 110 : dir_commit_chunk

디렉터리 엔트리 page를 write 한다.

line 126 : dir_get_page()

디렉터리가 포함되어 있는 page를 가져온다.

line 160 : sfs_find_entry()

한 디렉터리에서 요구된 항목의 엔트리가 있는 지 찾는다.

line 185 : sfs_add_link()

디렉터리에 항목을 추가한다.

line 226 : sfs_delete_entry()

디렉터리에서 항목을 지운다.

line 251 : sfs_make_empty()

디렉터리에서 모든 항목을 지운다.

line 289 : sfs_empty_dir()

디렉터리에 항목이 없는지 검사한다.

line 314 : sfs_inode_by_name()

이름으로 아이노드 번호를 찾는다.

- itree.c

line 25 : sfs_find_blk()

블록 번호을 찾는다.

line 43 : get_block()

READ / WRITE에 따라 블록 번호를 반환하거나 할당한다.

3.4 파일시스템 템플릿 테스트

파일시스템 템플릿은 완전히 동작을 하지만 성능의 개선을 위해 만들어지지 않았다. 템플릿을 통한 확장을 목표로 하기 때문에 현재 가지고 있는

제한사항 하에서 파일시스템의 필수 동작들을 제대로 수행하는 지가 테스트의 주안점이다.

3.4.1 shell에서 동작하는 파일 명령과의 상호작용

리눅스 쉘에서는 여러 가지 파일 명령들을 제공한다. 파일 명령들은 실제 파일시스템들에 시스템콜을 통한 요청을 한다. 유저 레벨에서 파일시스템의 동작을 가장 가시적으로 볼 수 있는 테스트이다.

테스트 대상이 되는 쉘 명령

mount / umount : 파일시스템 마운트/언마운트

ls : directrory list 를 보여줌. 최초의 마운트 후 각 객체들이 잘 연결이 되어 수행되는 지를 확인할 수 있다.

cp : 파일 카피를 하고, read / wrie의 동작 상태를 확인할 수 있다.

mkdir : 디렉터리 생성,

rmdir : 디렉터리 삭제

3.4.2 파일시스템 관련 system call 테스트

좀 더 세부적인 동작 유무를 확인하기 위하여 각종 시스템 콜을 테스트한다. 리컨값을 확인한다. READ / WRITE의 경우에는 실제 읽거나 쓰여진 데이터를 확인한다.

System Call	Object	Support
mount()	FILESYSTEM	YES
umount()	FILESYSTEM	YES
mkdir()	DIRECTORY	YES
chdir()	DIRECTORY	YES
opendir()	DIRECTORY	YES
readdir()	DIRECTORY	YES

closedir()	DIRECTORY	YES
rmdir()	DIRECTORY	YES
creat()	FILE	YES
open()	FILE	YES
write()	FILE	YES
lseek()	FILE	YES
read()	FILE	YES
close()	FILE	YES

표 3.13 파일시스템 템플릿의 시스템콜 지원유무 결과

제 4 장 결과 및 토의

4.1 결과 분석

리눅스 파일시스템을 통한 문서화를 하고, 그 지식을 기초로 하여 파일시스템 템플릿 즉 작고 간단한 파일시스템을 구현하고 테스트 하였다. 시스템 콜의 요청을 파일시스템 템플릿에서 성공적으로 처리하는 것을 확인하였다. 이것은 VFS의 인터페이스를 어떻게 접근하고 그것에 맞추어 구현하여야 하는 지를 설명한다. 파일시스템의 개발의 시작점이 될 수 있는 코드와 문서화로 파일시스템 개발의 기술 장벽을 낮출 수 있다.

4.2 템플릿 적용 방법

템플릿의 적용 다시 말해 리눅스 파일 개발 방법을 설명하려 한다. 다음의 3 단계로 진행한다.

4.2.1 파일시스템 설계

파일시스템 설계 단계에서 고려하여야 할 점은 다음과 같다.

- 파일시스템의 목적이 무엇인가?
- 그 목적을 위해 메타데이터를 어떻게 설계할 것인가?
- 데이터 블록을 어떻게 관리할 것인가?

파일시스템은 간단히 말해 데이터 블록을 어떻게 관리할 지에 대한 전략이다. 그 전략에 따라 메타데이터를 설계한다. 그래서 파일시스템의 주된 목적이 무엇인지를 결정하고 나면 파일(데이터블록)을 어떻게 관리할 지를 결정한다. 그 이후에 메타데이터를 적절히 배치하면 파일시스템의 디스크 레이아웃이 된다. 본 템플릿에서는 아주 간결한 레이아웃으로 그 예를 보였다. 실제 개발에 있어서는 더 복잡한 모습이 될 수도 있고 더 간단한 모습이 될 수도 있다. 그 실용적인 목적에 따라 디스크 레이아웃을 설

계하고 mkfs 프로그램을 제작하여 디스크에 파일시스템 레이아웃을 쓴다. 그것을 확인하기 위해 fsdb 프로그램을 작성하여 디스크 레이아웃이 적합하게 쓰였는 지를 확인한다. 응용 프로그램 레벨에서 확인하는 것은 중요한데 만일 커널 모듈(파일시스템)에서 확인하려 한다면 2단계의 검증이 필요하므로 파일시스템에서 이 레이아웃을 이용하기 전에 응용단계에서의 확실한 검증이 필요하다. 이 단계가 끝나면 파일시스템의 설계가 끝나고 디스크 상의 파일시스템은 사용가능한 상태가 된다. 이 응용 프로그램은 파일단위의 입출력이 아니고 device 파일을 직접 제어 하는 것이므로 이전의 파일시스템이 존재하였다면 파괴될 것이다. 그러므로 실험단계에서는 UML에서 다양한 테스트를 하도록 하고 그 테스트가 완전하다고 판단 될 시에 실제 디바이스에 파일시스템의 레이아웃을 쓰는 단계로 한다면 안전한 실험이 될 것이다.

4.2.2 파일시스템 구현

이 부분은 구현할 파일시스템에 따라 다르다. 공통적으로 구현해야 할 부분의 VFS의 인터페이스인데 여기서 본 파일시스템 템플릿을 사용할 수 있다. 파일시스템에 따라 다를 수 있지만 레이아웃이 비슷하다면 그대로 적용할 수 있는 부분은 많아진다. 하지만 레이아웃은 그 목적에 따라 다를 수 밖에 없고 최소한의 템플릿 사용은 객체와 오퍼레이션의 프로토타입과 그 안에서 사용된 커널API이다. 최소한의 사용이긴 하지만 VFS의 인터페이스를 작성하는 부분은 명확하기 때문에 생산성 향상을 가져온다. 공통의 인터페이스를 다 작성하고 나면 몇 가지 테스트가 필요하다. 모듈의 등록과 해제, 파일시스템의 마운트를 통해 공통의 인터페이스의 일부를 테스트한다. 이 테스트가 검증되면 파일시스템의 특화된 부분 즉 각 파일시스템의 전략에 따라 다른 부분들을 구현한다. 이 부분이 파일시스템 개발의 핵심이 되고 파일시스템 개발자는 이것에 초점을 맞춰야 할 것 이다.

4.2.3 파일시스템 테스트

파일시스템의 테스트는 기능 테스트와 성능 테스트로 나뉜다. 기능 테스트는 본 논문에서 실시한 테스트를 기본으로 하여 파일시스템에 따라 그 목록을 가감할 수 있다. 이 기본 테스트를 통해 파일시스템이 리눅스와 완벽히 호환되는 지를 검증할 수 있고 시스템 콜을 통해 파일시스템에 대한 요청이 가능한 지를 검증할 수 있다.

이 기본 테스트 이후에 이루어져야할 테스트는 성능 테스트 이다. 파일시스템이 지향하는 목표를 달성하는 가를 테스트 하는 것이다. 예를 들어 read에 편중에 I/O를 목표로 한 파일시스템의 경우(멀티미디어 디바이스) 라면 read 속도를 다른 파일시스템들과 비교를 하면 될 것이다. 이 성능 테스트를 통해 파일시스템에서 생기는 병목 현상을 찾아내고 개선하여 원하는 목적의 파일시스템을 만들어 낸다.

제 5 장 결론 및 향후 연구

5.1 결론

파일시스템은 운영체제에서 빼놓을 수 없는 중요한 역할을 하고 있고 그 복잡도 또한 증가하고 있다. 다양한 멀티미디어 디바이스가 출현함에 따라 파일시스템 역시 특정 목적에 적합한 형태로 구현되어야 한다. 리눅스는 이런 다양한 파일시스템 지원에 적합한 운영체제이나 그 구현에는 여러 가지 어려움으로 기술 장벽이 있다. 따라서 국내의 연구들은 리눅스에서 동작하는 별도의 모듈을 작성하여 그 목적을 달성하고 있으나 자원의 효율성, 호환성 측면에서 비생산적인 방법이다. 따라서 리눅스 파일시스템 개발의 기술 장벽이 되는 구현에 관한 문서에 부재를 문서화와 실제 간단한 파일시스템의 코드를 통해 낮추는 과정이 필요하다.

본 연구에서는 문서와 코드를 제공하고 또 파일시스템의 개발 방법을 제공하여 리눅스 파일시스템의 기술 장벽을 낮추고자 한다. 파일시스템에 관한 연구를 통해 개발에 필요한 모든 사항을 문서화 하였고 파일시스템이 공통적으로 구현해야할 부분을 구분하여 작은 파일시스템 템플릿을 구현하였다. 그러므로 본 논문과 파일시스템 템플릿을 가지고 파일시스템 개발시 필요한 사전작업의 대다수를 피할 수 있으며, 파일시스템 자체의 전략에만 집중할 수 있게 하였다.

5.2 향후 연구

본 템플릿 코드에서 보완하여야 할 사항은 첫째, 완전히 디스크 레이아웃에 투명한 템플릿으로 전환이다. 본 연구에서는 디스크 디바이스와의 상호작용을 밝히기 위하여 디스크 레이아웃에 기반한 파일시스템 템플릿을

구현하였고 실제 동작을 테스트 하기 위하여 실제 동작하는 파일시스템을 구현하였다. 연구를 통해 동작에 대한 검증이 완료되었으므로 동작에 관련 없이 파일시스템이 리눅스 VFS의 하위 모듈로 동작하기 위해 필요한 공통의 코드만으로 작성된 템플릿으로 전환이 필요하다. 이것을 통해 파일시스템 개발자는 함수의 접두어를 그 파일시스템으로 바꾸기만 하면 코드를 그대로 적용할 수 있을 것 이다. 둘째, 실용적인 파일시스템의 개발이다. 본 템플릿을 통해 파일시스템을 개발하되, 현재 가지고 파일시스템의 제약사항을 해결하고 특정 목적에 부합하는 파일시스템을 개발하는 것이다. 이것을 통해 파일시스템 템플릿과 문서를 좀 더 정제할 수 있을 것이다. 따라서 파일시스템 구현의 복잡도가 높아짐에 따라 문서 역시 복잡도가 높아지고 실용성은 더욱 높아질 것이다.

참 고 문 헌

[1] Bar, Moshe, Linux File Systems, McGraw-Hill Companies, 2001.

[2] Daniel Plerre Bovet, Marco Cesati, Understanding the Linux Kernel(3/E), O'REILLY, 2005

[3] Card, R., Ts'o, T., and Tweedie, S., "Design and Implementation of the Second Extended Filesystem", First Dutch International Symposium on Linux, 1994.

[4] Richard Grooch, Overview of the Linux Virtual File System, 1999

[5] 권우일, 윤미현, 이동준, 장재혁, 양승민, DVR 시스템을 위한 저널링 파일 시스템의 성능평가, 2002 추계 한국정보과학회 논문지, 2002.

[6] XFS html document of SGI, http://oss.sgi.com/projects/xfs/

[7] 원유집, 박진연, "정보가전용 멀티미디어 파일 시스템 기술" 멀티미디어와 정보화 사회, 한국과학기술진흥재단, http://211.40.179.13/book_file/ke28/ke028-index.htm.

[8] 원유집 외, "멀티미디어 스트림의 QoS를 보장하는 통합형 파일시스템", 한국 정보 과학회, 2000

[9] 이민석, "임베디드 멀티미디어 시스템을 위하 파일 시스템의 설계 및 구현", A Files Sytem for Embedded Multimedia Systems, 2004

[10] 이민석, "대형 멀티미디어 파일을 위한 파일 시스템 구현", Journal of Informaion Technolgy Application & Management, vol.10, No. 4, pp. 169-183, 2003.

[11] 리눅스 커널 소스 공식 사이트 www.kernel.org

개발 관련 참조 사이트

[1] 커널초보자를 위한 사이트 http://www.kernelnewbies.org/

[2] Kernel API http://lwn.net/Articles/2.6-kernel-api/

[3] Linux Kernel Module Programming Guide
http://www.tldp.org/LDP/lkmpg/2.6/html/

[4] 리눅스 커널 소스 공식 사이트 www.kernel.org

[5] 리눅스 심포지움 www.linuxsymposium.org

[6] Linux Filesystem Hirerarchy
tldp.org/LDP/Linux-Filesystem-Hierarchy/html/index.html

ABSTRACT

Research on the template for development of the Filesystem in Linux

Jang, Myoungwoo
Major in Computer Engineering
Dept. of Computer Engineering
Graduate School
Hansung University

A filesystem that store and retrieve data is a subsystem of Operating System. Although traditionally it is made for general purpose, a specific filesystem is necessary as multimedia devices emerge. linux which is developed under open source supports various filesystems providing a virtual filesystem layer. This layer makes a new filesystem be easily adapted, but it still difficult to write new one because of lack of documents.

It is said that there are not enough documents about writing a filesystem. It is needed to overcome difficulties of developing a filesystem. I documented the information related to Linux filesystem and wrote a file system templete by analyzing Linux filesystem. Researchers who want to write a filesystem might make it efficient with the thesis and the filesystem template.

부 록

1. 파일시스템 템플릿 소스 코드
2. 파일시스템 초기화 프로그램 소스 코드(mkfs.sfs)